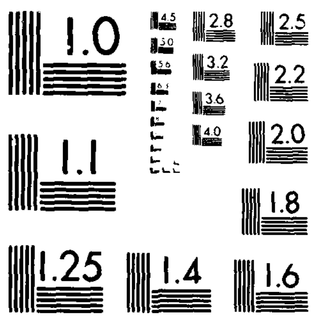
AD-A091 760   SOFTECH INC WALTHAM MASS   F/G 9/2
ADA COMPILER VALIDATION IMPLEMENTERS' GUIDE.(U)
OCT 80 J B GOODENOUGH   MDA903-79-C-0687
UNCLASSIFIED 1067-2.3   NL

1 OF 4
AD
A091760



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

ADA COMPILER VALIDATION
IMPLEMENTERS' GUIDE

1067-2.3

October 1, 1980

SOFTECH

THE SOFTWARE TECHNOLOGY COMPANY

ADA COMPILER VALIDATION IMPLEMENTERS' GUIDE

1067-2.3

October 1, 1980

Data Item A0004

Submitted to

Defense Research Projects Agency
Arlington, Virginia

Contract MDA903-79-C-0687

Prepared by

SofTech, Inc.
460 Totten Pond Road
Waltham, MA 02154

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER (14) 1067-2.3	2. GOVT ACCESSION NO. AD-A091	3. RECIPIENT'S CATALOG NUMBER 760
4. TITLE (and Subtitle) (6) Ada Compiler Validation Implementers' Guide		5. TYPE OF REPORT & PERIOD COVERED TR
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) (10) John B. Goodenough		8. CONTRACT OR GRANT NUMBER(s) (15) MDA903-79-C-0687
9. PERFORMING ORGANIZATION NAME AND ADDRESS The Software Technology Company (SOFTECH), Inc. 460 Totten Pond Road Waltham, MA 02154		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS (11) 1 Oct 80
11. CONTROLLING OFFICE NAME AND ADDRESS DARPA/IPTO 1400 Wilson Blvd. Arlington, VA 22209 (12) 307		12. REPORT DATE October 1, 1980
		13. NUMBER OF PAGES 311
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Same as 11.		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

LEVEL II

16. DISTRIBUTION STATEMENT (of this Report)

APPROVED FOR PUBLIC RELEASE
DISTRIBUTION UNLIMITED

DTIC ELECTE NOV 7 1980 S C

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Ada Compiler Validation
Ada Language
Computers
Implementers' Guide

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report is one of the documents being produced as part of the Ada Compiler Validation Capability (ACVC). The primary purpose of the ACVC is to help in deciding whether Ada translators conform to the Ada language standard. A secondary purpose is to help Ada implementers comply with the Ada standard.

The ACVC has three main components:

- An Implementers' Guide describing implementation implications of the Ada Standard and the conditions to be checked by validation tests.
- Test programs to be submitted to a compiler

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

405 73

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Item #20 -

- Validation support tools that assist in preparing tests for execution and in analyzing the results of executions.

More information about the ACVC can be found in "The Ada Compiler Validation Capability," a paper to be published in the proceedings of a Symposium on the Ada Programming Language, December 1980. Preprints are available from SofTech.

PREFACE

This document was produced under the supervision of John B. Goodenough, who is also the principal author. Other contributors were John R. Kelly, N. Lomuto, and Robert Mandl of SofTech, and Brian A. Wichmann of the National Physical Laboratory, England.

TABLE OF CONTENTS

Table of Contents

CHAPTER 1

Introduction

This report is one of the documents being produced as part of the Ada Compiler Validation Capability (ACVC). The primary purpose of the ACVC is to help in deciding whether Ada translators conform to the Ada language standard. A secondary purpose is to help Ada implementers comply with the Ada Standard.

The ACVC has three main components:

- An Implementers' Guide describing implementation implications of the Ada Standard and the conditions to be checked by validation tests.
- Test programs to be submitted to a compiler.
- Validation support tools that assist in preparing tests for execution and in analyzing the results of executions.

More information about the ACVC can be found in "The Ada Compiler Validation Capability," a paper to be published in the proceedings of a Symposium on the Ada Programming Language, December 1980. Preprints are available from SofTech.

This document is the Implementers' Guide. It follows the structure of the Proposed Standard Ada Language Reference Manual (LRM), July 1980. In case of conflict between the Implementers' Guide and the Standard, the Standard, of course, takes precedence.

Although the organization of this report follows the structure and numbering of the Ada Standard, we have sometimes divided the presentation into related subtopics. Such subsections are designated with letters, e.g., 8.3.a, 8.3.b, etc. to distinguish them from subsections of the Standard.

Each numbered/lettered Implementers' Guide section contains up to five unnumbered subsections:

1. Semantic Ramifications -- This subsection documents semantics implications of special interest to implementers that might not otherwise be obvious from a reading of the language definition. When conclusions stated in these subsections are derived from statements in separate sections of the Ada Standard, appropriate references are made so the basis for our interpretation is clear. When necessary, additional explanation for our interpretation will be given to aid those who are not intimately familiar with the Standard.

2. Compile-time Constraints -- This section will explicitly list context-sensitive syntactic and semantic constraints to be checked by an Ada translator prior to beginning execution of Ada programs. In essence, all legality constraints not expressed by BNF

productions in the Standard will be listed here, even if this listing duplicates wording in the Standard. The purpose of this subsection is to provide a checklist for implementers to ensure they cover all the context-sensitive validity constraints stated or implied by the Standard. Non-straightforward ways of violating these constraints will be discussed in the Semantic Ramifications subsection.

3. Exception Conditions -- This section explicitly states the conditions under which an implementation is required to raise an exception associated with some pre-defined language construct. The purpose of this subsection is to provide a consolidated checklist detailing the situations in which an implementation must raise an exception. In many cases, the content of this subsection will duplicate the content of the Standard, but when necessary, we will provide a more detailed exposition of situations giving rise to an exception than is provided by the Standard, since our goal is to help ensure that implementers do not overlook implications of the Standard and to ensure that our validation tests cover all these situations.

   In addition, we describe how the SUPPRESS pragma (see 11.7) is (probably) to be used to suppress the required checks. Since the description of this pragma in 11.7 of the LRM is fairly vague, we consider each of our assertions about how the pragma is to be used as a "Gap" (see below) to be resolved in the final version of the Standard.

4. Test Objectives and Design Guidelines -- This subsection specifies the validation tests to be written, lists problems to be kept in mind while writing test cases under "Implementation Guidelines," and when necessary, outlines the program structure required to satisfy a test objective. At least one test will be written for each objective, except when the objective is duplicated by another objective in another section. In such cases, a reference to the implemented objective is given.

   At this time, Test Objectives are numbered consecutively, but as this document grows and as Ada matures, it will be necessary to revise some objectives or add new ones whose position for expository purposes is not reflected by the numbering sequence. Hence, it is not a requirement that test objectives be arranged in numeric order, nor that the set of Test Objective numbers be dense. To prevent confusion with different test set versions, we never replace a Test Objective for an existing test with an identically numbered objective that is substantially different in its intent.

5. Gaps -- This subsection documents those aspects of the Ada standard that are unclear or inconsistent. In addition, when the stated rules do not seem consistent with the probable intent of the designers, we explain the possible inconsistency as a warning

that a change to the Proposed Standard might occur in this area. Since the current LRM is only a proposed Standard, we give our interpretation of how a Gap should be resolved; semantic ramifications, constraints, exceptions, or tests that depend on our resolution of a Gap are marked "(see Gaps)" to indicate that such IG content cannot be derived from current statements in the Standard. Our interpretations, of course, have no official standing. They merely reflect our opinion of how a problem will be resolved when the final version of te Standard is issued. We give these interpretations in the interest of encouraging the development of standard compilers even where the specification is ambiguous. When the Gaps are resolved officially, the IG will be updated appropriately.

We also document in this subsection any expected difficulties in constructing tests to satisfy the objectives.

This version of the Implementers' Guide corrects errors in the previous version, based on comments we have received. We have completed Chapters 8 and 11, added Chapters 9, 10, and 12, and substantially augmented Chapters 3 and 4. We have also added as an Appendix a cross reference listing to the syntax productions in the Standard. Comments on this version should be sent to John B. Goodenough at SofTech, or (for those with ARPANET access) to Goodenough @ ISI.

CHAPTER 2

Lexical Elements

2.1 Character Set

Interactions with other lexical sections are discussed in those sections.

Test Objectives and Design Guidelines

1. Check that the basic character set is accepted outside of string literals and comments.

2. Determine what extended characters are accepted, especially whether full ASCII or EBCDIC.

3. Check that lower and upper case letters are equivalent outside of string literals (see 2.3.T/1 and 2.4.T/2) and distinct within string literals (see 2.6.T/8).

2.2 Lexical Units and Spacing Conventions

Note that '--' is not considered a compound symbol (i.e., a lexical unit) since it introduces a comment, which is itself a lexical unit. Similarly, the double quote character is not considered a delimiter since it is part of a string literal.

The lexical processing of the apostrophe is complicated by the three contexts in which it is used: character literals, attribute naming, and qualified expressions, e.g.,:

```
if'('in'a'..')'then
if'('=')'then    -- two character literals, '(' and ')'
ASCII'('a')      -- 'a' qualified by ASCII
T'BASE'FIRST     -- attribute naming
```

To resolve the potential lexical ambiguities implied here, note that the sequence "apostrophe, character, non-apostrophe" always means the first apostrophe is a lexeme (if it is already known not to be the terminating apostrophe for a character literal). If the sequence "apostrophe, character, apostrophe" is seen, and the preceding lexeme is an identifier, the sequence should be considered a character literal if the preceding identifier is a reserved word. If it is not a reserved word, the first apostrophe should be considered a lexical unit. In short, to perform lexical analysis with at most two character lookahead, you must sometimes know whether or not the preceding lexeme was a reserved word.

Compile-time Constraints

1. The only ASCII control characters permitted in Ada programs are CR, LF, VT, HT, FF, DEL, and NUL.

2. No lexical unit can extend over a line boundary (see 2.3.C/1, 2.4.C/1, 2.4.1.C/1, 2.5.C/1, 2.6.C/1.

Test Objectives and Design Guidelines

1. Check that an identifier, reserved word, compound symbol, numeric literal, string literal, or comment cannot be continued across a line boundary.

   Implementation Guideline: This test is primarily intended to check that implementations accepting a fixed length input record do not concatenate records, but instead treat end-of-record as a separator character. Hence, the test should be coded to ensure the attempt to continue a construct across a "line" boundary occurs at an input record boundary. Since input record lengths will vary from one implementation to the next, this test must be parameterized to accept different input record lengths. For implementations that accept variable length input records up to a maximum length, the "continued" construct must appear at the end of a maximum length input record.

2. Check that the delimiters are accepted, especially the compound symbols.

3. Check that at least one space must separate adjacent identifiers (including reserved words) and/or numbers.

4. Check that lexical units such as identifiers (including reserved words), numbers, and compound symbols cannot contain spaces.

5. Check that all ASCII control characters other than carriage return, line feed, vertical tabulate, horizontal tabulate, form feed, delete, and null cannot appear in Ada programs.

6. Check that any of carriage return, line feed, vertical tab, and form feed are permitted outside string literals and are considered to signal the end of a line.

   Implementation Guideline: Try sequences such as LF, CR, CR or a single CR as line terminators as well as more customary sequences.

   Check that horizonal tabulate is allowed in comments and between lexical units, where it is considered equivalent to a space.

   Check that delete and null characters are ignored in all lexical units (see Gaps).

Gaps

1. The statement that "delete and null characters are ignored" <u>between</u> lexical units does not state that DEL and NUL should be ignored wherever they occur. As it is, a NUL or DEL between two characters, e.g., A and B, can only be ignored if the characters are part of different lexical units, but ignoring the NUL would imply A and B are part of a single lexical unit. If NUL serves to separate lexical units, is it being ignored? Moreover, is it really the intent to forbid NUL and DEL in comments (since certainly their occurrence there would not be "between lexical units"). We suspect the intent is to ignore DEL and NUL wherever they occur.

2.3 Identifiers

Semantic Ramifications

The rule stating that upper and lower case letters are equivalent is in LRM 2.1.

Note that even "external" identifiers such as names of library units (see 10.1) can be as long as the maximum input line length. Such names might exceed the length permitted by standard linkers in the compiler's operating system environnment. In such a case, an Ada implementation must provide a special linker.

Compile-time Constraints

1. No identifier can be longer than the maximum input line length permitted by an implmentation (see 2.2).

Test Objectives and Design Guidelines

1. Check that upper and lower case letters are equivalent in identifiers (including reserved words). (See also 8.3.d.T/2.)

   Implementation Guideline: Try some all-upper, all-lower, and mixed case identifiers.

2. Check that consecutive, leading, and/or trailing underscores are not permitted in identifiers.

3. Check that identifiers can be as long as the maximum input line length permitted by the implementation and that all characters are significant (e.g., not just the first 8 or 16, or not just the first m and last n characters). Try identifiers serving as variables, enumeration literals, subprogram names, parameter names, entry names, record component names, type names, package names (both library units and subunits), statement labels, block labels, loop labels, task names, and exception names.

   Implementation Guideline: Maximum length subprogram names and package names should be checked in separate tests.

4. Check that ? $ @ # ' are not permitted in identifiers.

2.4 Numeric Literals

Semantic Ramifications

The type of a number (e.g., INTEGER vs. LONG_INTEGER or FLOAT vs. fixed) is contextually determined, see 4.2 (and wherever overloading of literals is discussed).

Compile-time Constraints

1. No decimal_number can contain more characters than the maximum input line length permitted by an implementation (see 2.2).

Test Objectives and Design Guidelines

1. Check that underscores are not permitted to:

   a. be consecutive within a number,

   b. lead or trail in a number,

   c. be adjacent to (on either side) the "#", ":", ".", "E", "e", "+", "-" characters in a number.

2. Check that "E" and "e" may be used in decimal_numbers and based_numbers (for those compilers that support upper and lower case).

3. Check that leading zeros in numbers (and in integral subparts such as bases and exponents) are ignored (i.e., no errors or warnings, get correct value). Check that trailing zeros in the fractional part of decimal_numbers are ignored. Check that numbers can be as long as the maximum input line length permitted by an implementation (and yield representable values, so use leading/trailing zeros).

4. Check how many significant digits (excluding trailing zeros) in the fractional part of decimal_numbers are used to form the value. Check it for fixed and float types.

5. Check that leading/trailing decimal points are not permitted in decimal_numbers.

2.4.1 Based Numbers

Compile-time Constraints

1. No based number can contain more characters than the maximum input line length permitted by an implementation (see 2.2).

2. The extended digits comprising a based number must all have a value less than the value of the base (see Gaps).

Test Objectives and Design Guidelines

1. Check that a based_integer always yields a nonnegative value.

2. Check that non-consecutive embedded underscores are permitted in every part of a based_integer.

3. Check that based_integers and based_numbers with bases 2 through 16 all yield correct values.

4. Check that the digits and extended_digits of a number are within the correct range for the number's base (see Gaps).

Gaps

1. The LRM does not actually restsict the digits of a based number to those whose value is less than the base.

2.4.1.a Based Real numbers

Semantic Ramifications

Model real numbers (see 3.5.6-9) can be written with a base that is a power of 2. Since model numbers must be represented exactly, such based numbers must be converted exactly to the appropriate bit pattern. Hence, the use of a binary radix to hold numbers internally in the compiler is recommended. However, in the case of a fixed point value of a type with a representation specification, model numbers are not necessarily the values expressed with a mantissa and binary radix. For instance, 0.1 will be a model number if ACTUAL_DELTA = 0.01. The simplest method to overcome this problem is to always round numbers correctly to the corresponding type and keep literals internally with a high precision (see 4.10 for a further discussion).

Test Objectives and Design Guidelines

11. Check real literals with bases from 2 to 16. (Use corresponding check for integers by inserting ".0" suitably.

12. Check that initial zeros do not affect the accuracy of values.

Gaps

1. It is possible to represent the same literal value in different ways in the source text of a program, for instance, 0.01E1 and 0.1, 2#0.01# and 0.25, 16#0.F# and 4#0.33#. It is not clear in the case of real values which do not represent model numbers in the executed program if these pairs should give the same value. The manual would appear to allow the freedom to give different values, although this would lead to some

surprises for the programmer. Even in the case of the single representation of 0.1, its use could depend upon whether the compiler required the value in an overlength accumulator or in memory (with fewer bits).

2.5 Character Literals

Semantic Ramifications

The type of a character literal is contextually determined (see 3.5.1, 4.2, and 4.7).

Compile-time Constraints

1. A character literal cannot extend across a line boundary (see 2.2).

2. The character enclosed in apostrophes must be one of the 95 ASCII graphic characters, including space.

Test Objectives and Design Guidelines

1. Check that all standard ASCII character literals have the correct representation.

2. Check that non-graphic characters are not allowed (see 2.2).

3. Check that forms like, if'b'in'a'..'c' and T'BASE'FIRST are lexically analyzed correctly.

2.6 Character Strings

Semantic Ramifications

Note that string literals start with index 1 (see 3.6.3), including null strings. Note that the " character must be doubled to include it in strings.

The type/subtype of a string or character literal is contextually determined (see 3.5.1, 4.2, and 4.6). A string literal is a shorthand way of writing a positional array aggregate of character literals (which are overloadable enumeration literals) (see 3.6.3). Consequently, string literals are never padded in assignment or comparison contexts.

Compile-time Constraints

1. A character string cannot extend across a line boundary (see 2.2).

2. Non-graphic ASCII characters are not permitted in string literals.

Test Objectives and Design Guidelines

1. Check that string literals started with " must end with ".

2. Check that " must be doubled to be used within string literals bracketted with this character.

3. Check that string literals cannot cross line boundaries (see 2.2.T/1).

4. Check that all printable characters are permitted in string literals.

   Implementation Guideline: Check separately for the basic characters (see 2.1) and any extended characters the implementation accepts. (Note that cent-sign is in EBCDIC but not in ASCII.)

5. Check that non-printing (control) characters (other than space) cannot be included in string literals. In other words, the & operator and the enumeration literals (e.g., HT) must be used to build string values containing control characters.

6. Check that all ASCII characters, including control characters, can appear anywhere in string values. In particular, check that string values can include NUL and DEL. (Note that in the C language, NUL is used to mark the end of a string and so can't be included in strings.)

7. Generate a string with INTEGER'LAST characters.

8. Check that upper and lower case letters are distinct within string literals.

2.7 Comments

Test Objectives and Design Guidelines

1. Check that a comment is terminated by the end of the line, i.e., not by the next '--', whether on the same line or the next line. (See implementation guideline for 2.2.T/1.)

2. Check that legal Ada statements and pragmas contained in comments have no effects.

2.8 Pragmas

2.9 Reserved Words

Semantic Ramifications

Note that DIGITS, DELTA, and RANGE are reserved words used both in declarations and as predefined attributes (3.5.8, 3.5.10 and 3.6.2). The lack

of boldface type for the DIGITS and DELTA attributes does not imply these identifiers are unreserved.

Compile-time Constraints

1. Reserved words cannot be given user_defined meanings via declarations.

Test Objectives and Design Guidelines

1. Check that all the reserved words are actually reserved.

2. Check that all the reserved words and keywords in languages such as PL/I (IBM F and optimizing versions, MULTICS, standard, PL/C, etc.), Pascal, Jovial (J73, J73I, J73C, J3, J3B), Tacpol, CMS-2, SPL/I, COBOL, FORTRAN 77, Algol 60, Algol 68 that are not reserved words in Ada are actually not reserved.

3. Check that all predefined attributes (except DIGITS and DELTA) and all predefined type and package names are not reserved.

2.10 Transliteration

Test Objectives and Design Guidelines

1. Check if the replacements ! for |, % for ", and : for # are accepted.

   Check that the replacements for # and " must be consistent within a lexical unit.

2. Check that a string literal delimited by % must not contain ".

CHAPTER 3

Declarations and Types

3.1 Declarations

The statement that attributes are "predefined and cannot be declared" should not be interpreted to mean that attribute names are reserved or that variables cannot be declared with names identical to attribute names. The statement merely means there is no language-defined way of adding to the set of attributes that can be accessed using attribute notation (see 4.1.4).

Ramifications, constraints, exceptions, and tests associated with each form of declaration are presented in the sections where these forms are further defined.

3.2 Object and Number Declarations

3.2.a Object Declarations

Semantic Ramifications

Note that assignment is not available for limited private types (7.4.2), task types (9.1), and composite types having a component of a type for which assignment is not available. In addition, a constant of a private type cannot be initialized when it is declared in the visible part of a package (7.4). These are the only situations in which initialization expressions are forbidden in object declarations.

The significance of "introducing" the identifiers of a declaration is discussed further in 8.3.S.

If an explicit initialization is given for a record or array type, all components of the object must be given initial values (see 4.3). The only situation when only some components are given initial values is when default values are specified for some components of a record type (see 3.7). It is possible that a default initial value will not satisfy the constraints of the particular object declared; CONSTRAINT_ERROR will be raised when the default initialization is performed (see 3.7.S).

Note that except for deferred constants of private types, a constant object must be explicitly initialized -- no default initialization is permitted.

Default initial values are provided for variables of access types (see 3.8), task types (see 9.1). and for record types whose components have been declared with default initial values.

Details pertaining to the declaration of objects of various types are

given in the sections pertaining to these types. We are concerned here only with expressing constraints that apply in general to object declarations. In addition, issues concerning when objects may be declared with the same identifier are discussed thoroughly in 8.3.S.

Note that there are no restrictions limiting initialization expressions to static expressions. Variables and functions may be freely invoked in initialization expressions as long as the variables have been given a value and the bodies of the invoked functions have been elaborated (see 3.9). The effect of these restrictions is that functions declared in package specifications cannot be invoked in initialization expressions within the package specification (since bodies of subprograms cannot be given in package specifications; see 7.2).

Note that since a type definition introduces a unique type (see 3.3), A and B below have the same type and can be assigned to each other, but C cannot be assigned to A or B, since C has a different type introduced by its type definition.

```
A,B: array (1..5) of INTEGER;
  C: array (1..5) of INTEGER;
```

Compile-time Constraints

1. An initialization expression must not be given for objects of a limited private or task type, or composite objects having a component of a type for which assignment is not available.

2. Objects declarations containing the word constant must have an initialization expression, except when the object being declared is of a private or limited private type and the declaration appears in the visible part of a package specification.

3. An unconstrained array_type_definition is not permitted in object declarations.

4. An identifier declared in an object declaration must not have been declared in any preceding declaration of the same declarative part (see 8.3).

5. No two identifiers in an object declaration's identifier list can be identical.

Exceptions

CONSTRAINT_ERROR is raised when the initial value of an object (whether specified explicitly or by default) does not satisfy a constraint imposed on the object or when the subtype_indication or array_type_definition raises CONSTRAINT_ERROR. The situations giving rise to this exception are discussed in detail for each data type individually.

Test Objectives and Design Guidelines

Detailed tests of object_declarations are specified for each data type individually.

1. Check that objects of a limited private type cannot be given initial values, except in the private part of a package specification (see 7.4.2.T/).

2. Check that a task object cannot be given an initial value (see 9.1.T/?).

3. Check that constant object declarations must have an explicit initialization expression, even when a default value exists for every component of the object.

4. Check that unconstrained array_type_definitions are not permitted in object declarations (see 3.6.T/1).

5. Check that an object cannot be given a name previously declared in the same declaration part (see 8.3.T/?).

6. Check that if several identifiers are declared in the same object declarations with an array type definition, they all have the same type, but if the identifiers are declared in separate declarations with the same array type definition. they have different types.

3.2.b Number Declarations

Semantic Ramifications

A number declaration names a numeric value equal to that of the literal_expression given in the declaration. If the literal_expression is of type universal_integer, the name can be used in any context requiring an integer value, and is implicitly converted to the appropriate predefined or user-defined integer type (see 3.5.4). In essence, the name introduced by a number declaration acts as a macro for a literal having the value of the literal_expression.

Integer literals cannot be used in contexts requiring real literals and vice versa (since values of a universal integer type can only be converted to an integer type, and values of a universal real type can only be converted to a fixed or floating point type; see 3.5.6). Hence, the compiler must determine whether the literal expression denotes an integer or real value, since the type of the number name determines in what contexts it can later be used legally.

Note that many of the predefined attributes (see Appendix A and below) and all of the LRM-defined SYSTEM constants (see 13.7) are names denoting literal values and consequently, can be used in number declarations.

Compile-time Constraints

1. The initializing expression in a number_declaration must be a literal_expression.

2. An identifier declared in a number_declaration must not have been previously declared in the same declarative part (see 8.3).

3. No two identifiers in a number declaration's identifier list can be identical (see 8.3).

Test Objectives and Design Guidelines

1. Check that the following attributes cannot appear in a number declaration: 'ADDRESS, 'BASE, 'SIZE, 'FIRST, 'LIST, 'IMAGE, 'VALUE, 'POS, 'VAL, 'PRED, 'SUCC, 'MACHINE_ROUNDS, 'MACHINE_OVERFLOWS, 'LENGTH, 'RANGE, 'CONSTRAINED. 'POSITION, 'FIRST_BIT, 'LAST_BIT, 'STORAGE_SIZE, 'TERMINATED, 'COUNT.

   Check that a user-defined function, operator, or the operator "&" cannot appear in a number declaration.

   Check that a string literal or character literal cannot appear in a number declaration.

2. An integral number name cannot be used in a context requiring a real value, and a real number name cannot be used in a context requiring an integer value.

3. Check that the following attributes can appear in number declarations: 'DELTA, 'ACTUAL_DELTA, 'BITS, 'LARGE, 'DIGITS, 'MANTISSA, 'EMAX, 'SMALL, 'EPSILON, 'MACHINE_RADIX, 'MACHINE_MANTISSA, 'MACHINE_EMAX, 'MACHINE_EMIN, and 'PRIORITY.

   Check that the SYSTEM constants STORAGE_UNIT, MEMORY_SIZE, MIN_INT, and MAX_INT can appear in number declarations.

4. Check that an identifier in a number declaration cannot have been declared previously in the same declarative part or in the same declaration (see 8.3.T/?).

3.3 Type and Subtype Declarations

Semantic Ramifications

The distinction between a type and a subtype as used in the LRM is similar to the distinction between a set and a subset -- just as a subset is a set, so a subtype is a type, and just as a subset of a set can be equal to the set itself, so a subtype can be equivalent to a type. For example, any type can be renamed using a subtype declaration. The new name would generally still be called a type name. Similarly, a type declaration can introduce a subtype name, as is explained in 3.4 for derived types and in 3.5.4 and 3.5.7

for numeric types. In general, the term "subtype" is used when the LRM wishes to emphasize the constraints that are imposed on a type; the term "type" is used when talking about subtypes or types in general. Sometimes the LRM speaks of a "constrained type," e.g., for record and array types, instead of a "subtype". In short, the most accurate reading of the LRM is obtained if one consider the terms "type" and "subtype" to be essentially synonymous and considers the term "subtype" to suggest the presence of constraints, although strictly speaking, subtypes need not be constrained types.

The restriction forbidding recursive type declarations means declarations like the following are forbidden:

```
type T is record
    A: T;                                -- illegal
    B: array (1..10) of T;               -- illegal
    end record;
type U is array (1..10) of U;            -- illegal
type V is access V;                      -- illegal
type R is digits R'DIGITS;               -- illegal
type S is digits 5 range 0.0..R'LARGE;   -- illegal
```

Note that the following (useless) types can, however, be declared.

```
type W;
type X is access W;
type W is access X;
```

Compile-time Constraints

1. The identifier introduced by a type or subtype declaration must not have been previously introduced by a declaration in the same declarative part (see 8.3).

2. If a type_declaration contains a discriminant_part, the type_definition must be a record_type_definition, private_type_definition, or derived_type_definition (see 3.7.1).

3. A type_mark must be the name of an entity introduced by a type_declaration, subtype_declaration, incomplete_type_declaration, task type specification, or a generic formal parameter type declaration.

4. A subtype_indication can contain a range_constraint only if the type_mark denotes a scalar type (see 3.5.C for additional restrictions).

5. A subtype_indication can contain an accuracy_constraint only if the type_mark denotes a real type (see 3.5.7 and 3.5.9 for additional restrictions).

6. A subtype_indication can contain an index_constraint only if the type_mark denotes an unconstrained array type (see 3.6.1 for additional restrictions) or an access type designating an unconstrained array type (see 3.8).

7. A subtype_indication can only contain a discriminant_constraint if the type_mark denotes an unconstrained record or private type with discriminants (see 3.7.2 for additional restrictions) or an access type designating an unconstrained record or private type with discriminants (see 3.8).

8. The name of the record type being declared cannot be used as the type_mark and cannot be specified as the type of any of its components nor as the component type for an array component.

9. The name of the array type being declared cannot be specified as the type of the array's component.

10. The name of the access type being declared cannot be specified as the type_mark in the access_type_definition's subtype_indication.

11. The 'BASE attribute can only be followed by the name of another attribute suitable for the base type. (See subsequent sections for more detailed constraints).

Exceptions

Exceptions can be raised during the elaboration of discriminant_parts, type_definitions (except for enumeration and private_type_definitions), or the constraint in a subtype_indication. The situations giving rise to these exceptions are specified in the sections where each of these syntactic forms is further described.

Test Objectives and Design Guidelines

1. Check that a type_declaration, incomplete_type_declaration, or subtype_declaration cannot introduce an identifier introduced previously in the same declarative part.

2. Check that discriminant_part cannot be supplied when declaring an enumeration, integer, real, or array type.

3. Check that in a subtype_indication

   - a range_constraint is not permitted for array, record, access, or private types;

   - an accuracy_constraint is not permitted for enumeration, integer, array, record, access, or private types;

   - an index_constraint is not permitted for scalar, record, or private types.

   - an index_constraint is not permitted for a constrained array type (see 3.6.1.T/1);

   - an index_constraint is not permitted for an access type

designating a scalar, record, private, or constrained array type (see 3.8.T/).

- a discriminant_constraint is not permitted for scalar or array types.

- a discriminant_constraint is not permitted for constrained record types (see 3.7.1.T/1);

- a discriminant_constraint is not permitted for an access type designating a scalar, array, or constrained record type (see 3.8.T/).

Implementation Guideline: A subtype_indication is used in many different contexts. Not all these contexts need be checked in the above tests.

4. Check that T'BASE cannot be used as a type_mark in a subtype_declaration, object_declaration, discrete_range, qualified_expression, or type_ conversion.

3.4 Derived Type Definitions

Semantic Ramifications

The definition of a parent type means that given the subtype declaration:

```
subtype INT is INTEGER range 1..100;
```

then the derived type

```
type MY_INT is new INT;
```

has the parent type INTEGER, since INT'BASE = INTEGER. The base type of MY_INT is an unnameable (i.e., anonymous) integer type whose constraints are those of the base type of INT. Hence, MY_INT'BASE'FIRST and INT'BASE'FIRST have the same value, but their types are different (since the base types are different). Similarly, MY_INT'FIRST and INT'FIRST have the same value but different types, i.e., INTEGER(MY_INT'FIRST) = INT'FIRST.

The LRM says a derived type belongs to the same class of types as its parent type. The classes of types mentioned in the LRM are: scalar, discrete, enumeration, integer (including universal_integer), predefined integer (excluding universal_integer), real, floating, fixed, composite, array, record, access, private, limited private, and task types. In addition, there are types for which assignment is not defined, namely, task types, limited private types, and composite types having a component of a type for which assignment is not defined. Similarly, there are types for which equality is not defined: task types, limited private types for which equality has not been defined by a user, and composite types containing a component for which equality is not defined. Various rules in the language are defined as applying to these classes of types. Each of these rules therefore applies to a derived type whose parent belongs to one of these classes.

Note that since literals for a derived type are the same as for the parent type, literals are not restricted to the range explicitly given for a derived type. For example, if X has been declared as a MY_INT variable, the assignment:

```
X := 101;
```

is legal, but the assignment raises CONSTRAINT_ERROR, since 101 is outside the permitted range of MY_INT values. Note that 101 is not outside the range of MY_INT'BASE values, assuming 101 is less than INTEGER'LAST, and so X < 101 is legal and does not raise CONSTRAINT_ERROR.

The fact that derived subprograms are implicitly declared where the derived type definition occurs means these subprograms can be named using the same conventions that would apply to an explicit declaration. For example, given the package:

```
package P is
    subtype INT is INTEGER range 1..100;
    type MY_INT is new INT;
    A, B: MY_INT;
end P;
```

all the predefined integer operators are overloaded to take MY_INT operands and produce MY_MINT results, e.g., the "+" operator is implicitly declared in P as

```
function "+" (X, Y: MY_INT'BASE) return MY_INT'BASE;
```

and hence can be named as P."+". (Note : the above use of 'BASE is not actually permitted in Ada, (see 3.3), but since the base type of MY_MINT is unnameable we use the notation to indicate the effect described in the LRM.) Note also that "systematic replacement of the parent type" (in this case, the parent type is INT'BASE, i.e., INTEGER), means <u>all</u> occurrences of the parent type as an argument or result type are replaced.

Invocations of P."+" are replaced by invocations of STANDARD."+", since P."+" was derived from the predefined "+" operator. If A and B are MY_INT variables, then A + B is evaluated as:

```
MY_INT(STANDARD."+"(INTEGER(A), INTEGER(B))
```

Hence, A + B raises NUMERIC_ERROR if and only if STANDARD."+" would raise this exception, and it raises CONSTRAINT_ERROR if the sum is outside the range of MY_INT values. Note that 101-B raises no exception because 101 is a valid MY_INT'BASE value even if it is not a valid MY_INT value, and the result will lie in the range of MY_INT values.

The predefined "+" operator could have been replaced by a user-defined "+" operator, e.g., one that performed addition modulo 100. Other subprograms unique to MY_MINT objects could also be introduced:

```
package Q is
    type MY_INT is new INTEGER range 1..100;
    A: MY_INT;
    function "+"(X,Y: MY_INT) return MY_INT;
    function F(X: INTEGER; Y: MY_INT) return INTEGER;
    function F(X: INTEGER; Y: MY_INT) return MY_INT;
    function G(X: MY_INT range 1..50) return MY_INT range 51..100;
    type NEW_INT is new MY_INT;
    D: NEW_INT;
end Q;
```

Note that with the new "+" definition, A + 0 raises CONSTRAINT_ERROR because 0 is not in the range of MY_INT values and "+" has been defined to accept only MY_INT arguments. Note also that the functions, F, G, and the redefined "+" function are not derived for the NEW_INT type since NEW_INT was defined prior to the end of the package specification. Hence, if D is a NEW_INT variable, D + 0 raises no exceptions and D + 50 could raise CONSTRAINT_ERROR, because the result could exceed NEW_INT'LAST, whereas A + 50 would not raise CONSTRAINT_ERROR since the redefined "+" operator would be invoked (the one that performs addition modulo 100). Note also that MY_INT'BASE'FIRST has the same numeric value as INTEGER'FIRST.

Now suppose we define a new package:

```
with Q;
package R is
    type NEWER is new Q.MY_INT range 51..100;
    RD: NEWER := 51;
end R;
```

The following subprograms are derived by NEWER from the subprograms declared in Q (using the suggestive, but illegal, 'BASE notation, and recalling that the derived subprogram specifications replace the parent type (MY_INT'BASE) with the unconstrained newly derived type (i.e., NEWER'BASE). The original Q."+" function was specified using MY_INT as a subtype_indication. But MY_INT is really an abbreviation for MY_INT'BASE range 1..100. Hence, the original Q."+" is actually defined as:

```
function "+"(X, Y: MY_INT'BASE range 1..100)
        return MY_INT'BASE range 1..100);
```

Therefore, the "+" subprogram derived in R is implicitly declared as:

```
function "+" (X, Y: NEWER'BASE range 1..100)
        return  NEWER'BASE range 1..100);
```

Hence, RD + 2 will not raise CONSTRAINT_ERROR even though 2 is not a valid NEWER value. CONSTRAINT_ERROR is raised only if RD + 2 not in NEWER. (Recall that the result of the derived "+" is converted to a NEWER value, and so must satisfy NEWER's range constraints.)

The F functions are derived similarly:

```
function F(X: INTEGER; Y: NEWER'BASE range 1..100) return INTEGER;
function F(X: INTEGER; Y: NEWER'BASE range 1..100)
                   return NEWER'BASE range 1..100;
```

And finally, applying the same rules to G, we get:

```
function G(X: NEWER'BASE range 1..50)
              return NEWER'BASE range 51..100;
```

In this case, every call to G with a NEWER variable will raise CONSTRAINT_ERROR, since no NEWER value lies in the range 1..50. However, a call with a literal, e.g., G(5) would be acceptable.

Finally, suppose the original definition of G had specified a range bound with a non-literal value, e.g.,

```
function G(X: MY_INT range 1..A) return MY_INT range A+1..100;
```

Then the derived function in R is equivalent to:

```
function G(X: NEWER'BASE range 1..NEWER'BASE(Q.A))
       return NEWER'BASE range NEWER'BASE(Q.A+1..100);
```

where the value of Q.A that is used is the value that was used in the original definition of Q.G. We have not had to explicitly convert any of the literal values used in the previous declarations, since these are implicitly converted to the appropriate integer type (3.5.4).

Note also that if NEWER were declared in Q's package body, the same subprogram derivations would occur, since NEWER's declaration is given "after the end of the package specification."

Not all subprograms applicble to a derived type, T, can be further derived. In particular, only those subprograms declared in a package specification containing the declaration of T can be derived when T is used as the parent type in another derived type declaration (see Gaps). Hence, if the declarations in Q are instead given in, say, a package body, none of the explicitly declared subprograms, i.e., F, G, or the redefined "+" subprogram, will be derived by any type whose parent is MY_INT. The only subprograms derived by NEW_INT, for example, would be those subprograms that were <u>derived</u> for MY_INT, not those subprograms that were later <u>declared</u> to be applicable to MY_INT values (see Gaps).

It is important to note that <u>only</u> those subprograms declared in the same package where a derived type is declared can be inherited (i.e., derived) by another type for which T is the parent. For example, consider:

```
packge P is
    type T is new INTEGER;
    subprogram PP(X: T);
end P;
```

```
with P;
package Q is
    procedure RR(X: T);
end Q;

with P,Q;
package S is
    procedure SS(X: T);
    type U is new T;
```

Q.RR is not derived by the declaration of U. Only P.PP is derived together with the predefined operations for INTEGERs.

Note that the set of predefined operations differs for different types. Numeric types have all the arithmetic and relational operators. The predefined operators for arrays, in general, consist only of the equality and membership operators. But "&" is predefined for any one dimensional array type, so any type derived from such a type will also derive the "&" subprograms (see 4.5.3). Similarly, types derived from a one-dimensional array of BOOLEAN type will derive the and, or, xor and not operators.

Compile-time Constraints

The only constraints are those that are normally present for subtype indications, and these are expressed in detail for each type of subtype indication.

1. A type_declaration can only contain a discriminant_part if the type_definition is a record_type_definition or a private_type_definition.

Exceptions

The only exceptions are those raised by elaboration of a subtype indication, and these are described in detail for each type of subtype indication.

Test Objectives and Design Guidelines

1. Check that derived subprograms can be named using the same form of selected component name that would be appropriate for an explicitly declared program.

   Implementation Guideline: Check for derived types declared in package specifications and bodies, subprogram bodies, blocks, and tasks.

2. Check that (implicitly declared) derived subprograms can be redefined in the declarative part containing the derived type declaration.

   For a derived type, T, declared in a package specification, check that both explicitly and implicitly declared subprograms for T:

   - can be derived when T is used as a parent type outside the package specification;

Implementation Guideline: Try using T as the parent type in a declaration in the package body, in a separately compiled unit, and, when the package containing T's declaration is nested in a declarative part, in a declaration appearing later in the same declarative part.

- are not derived when T is used as a parent type within the same package in which T is declared;

  Implementation Guideline: Include a derivation in the private part of the package, and in a nested package.

3. For a derived type, T, not declared in a package specification, explicitly declare some subprograms applicable to T. Check that these explicitly declared subprograms are not derived by a later type declaration, U, for which T is the parent type, but that the (implicitly declared) subprograms derived by T are also derived by U.

4. Check that only those subprograms declared in the same package as a derived type, T, are derived when T is used in another package as a parent type (see the example using both package P and S).

5. Check that the appropriate set of predefined operations (and only these predefined operations) are derived for the various predefined types and classes of types:

   - for all types, the membership operator;
   - for an integer type, the arithmetic and relational operators, including operators on fixed point types ("*", "/", and "**") and floating point types ("**");
   - for a floating point type, the arithmetic and relational operators including "**", but excluding "mod" and "rem".
   - for a fixed point type, the arithmetic and relational operators (including operators on integer and fixed point types, namely, "*", "/", and "**"), but excluding "mod" and "rem".
   - for enumeration types, the relational operators;
   - for all arrays whose components have equality (pre)defined, the equality operators;
   - for one-dimensional arrays, the "&" operator subprograms (see 4.5.3).
   - for one-dimensional arrays of BOOLEAN, the and, or, xor, and not operators;
   - for one-dimensional arrays of a discrete type (including, of course, a type derived from a discrete type), the relational operators;

- for record types having no component for which equality is not available, the equality operators;
- for private (but not limited private) types, the equality operators;
- for limited private and task types, no predefined operators other than the membership operators.

6. Check that the result of invoking a derived function (such as "-" for MY_INT) must satisfy the range constraints of the derived type rather than the range contraints of the parent type or the derived type's base type.

Gaps

The paragraph after the bullets might be clearer if it were rephrased as follows:

> For a predefined [parent] type, the subprograms that are derived are the corresponding predefined operations. The subprograms derived by a derived type, [T, are] further derived if [T] is used as [the] parent type in another derived type definition. [In addition,] if [T] is declared in a package specification, the subprograms [that are] applicable to [T] and [that are] declared in the package specification are derived [when T is used as the parent type in another derived type definition, but only if this other] derived type definition is] given after the end of the package specification.

We believe that this rephrasing expresses the same intent as the current rephrasing, but makes clearer that the last sentence in the current paragraph gives an additional subprogram derivation capability that applies to subprograms that are explicitly declared, rather than derived, for a type, and only when the subprogram declarations occur in the package specification where T is declared.

3.5 Scalar Types

Semantic Ramifications

The term "integer type" includes all predefined integer types, e.g., INTEGER, SHORT_INTEGER, LONG_INTEGER, etc. It also includes types derived from an integer type (see LRM 3.4).

The term "enumeration type" includes any user-defined or predefined enumeration type, as well as types derived from enumeration types.

The term "real type" includes all the predefined floating point types, e.g., FLOAT, LONG_FLOAT, etc., as well as user-defined fixed and floating point types. It also includes types derived from real types.

Additional predefined attributes that apply to scalar types are defined

in LRM Appendix A. In particular, IMAGE(X) returns the string representation of X; when S is a string, VALUE(S) returns the scalar value of S. In addition, other attributes are defined for certain classes of scalar types. These attributes are discussed below in the appropriate section for each type class.

3.5.1 Enumeration Types

Semantic Ramifications

When the IMAGE attribute is applied to an enumeration identifier, the case of the enumeration value need not be preserved (see LRM 2.3). Hence, given

```
type COLOR is (Red, Green, Blue, Yellow);
```

COLOR'IMAGE(Red) could equal "RED" (e.g., if the target computer does not support lower case printable characters), and COLOR'VALUE("RED") = COLOR'VALUE("red") = Red.

Compile-time Constraints

1. Duplicate enumeration literals (ignoring possible differences in case) are not permitted in a given enumeration type definition (see LRM 8.3 and 8.3.d.C/1).

Gaps

1. Issues involving overloaded enumeration literals will be explored in more detail later.

2. Issues involving enumeration type representations, in particular, representations that do not form a dense set of integers, will be given attention in Chapter 13.

3. The type of T'POS's result is contextually determined. Tests depending on this information are not yet designed.

Test Objectives and Design Guidelines

1. Check that only identifiers and character literals (not string literals) are allowed as enumeration literals.

   Check that at least one enumeration literal is required (either an identifier or character_literal), and that exactly one is permitted.

   Check that enumeration literals can have the maximum length permitted for identifiers (see 2.3.T/3).

2. Check that an enumeration literal from one enumeration type may appear again in another enumeration type definition (see 8.3.d.T/1).

3. Check that duplicate enumeration literals (including character literals) are not permitted in a single enumeration type definition (see 8.3.d.T/2).

4. Check that the predefined attributes FIRST and LAST give correct results for types and subtypes.

   Implementation Guideline: Use a subtype S of T such that S'FIRST /= T'FIRST and S'LAST /= T'LAST.

3.5.2 Character Types

Semantic Ramifications

A character type is a subclass of enumeration types in that a character enumeration type must contain at least one character literal. The only semantic significance of character types is that their character literals can be used in character string literals (see LRM 3.6.3). For example, given:

```
type OCTAL_CHAR is ('0', '1', '2', '3', '4', '5', '6', SEVEN);
type OCTAL_STRING is array (NATURAL range <>) of OCTAL_CHAR;
X: OCTAL_STRING(1..5);
```

X := "11664" assigns an OCTAL_STRING value to X (since "11664" is equivalent to the array aggregate ('1', '1', '6', '6', '4')), and X & SEVEN yields a valid OCTAL_STRING value (see LRM 4.5.3).

Test Objectives and Design Guidelines

1. Check that string literals are permitted for user-defined character types, and that concatenation is defined between strings and enumeration literals of the type.

   Check that the use of character literals not in the user-defined type is forbidden, e.g., for the type OCTAL_CHAR, that X := "11774" is illegal, since '7' does not appear in the definition of OCTAL_CHAR. Check for embedded blanks or underscores, when they are not in the type.

3.5.3 Boolean Type

Semantic Ramifications

Boolean types include those enumeration types derived from the predefined BOOLEAN type (see LRM 3.4). However, derived Boolean types cannot be used where the language requires expressions only of the predefined BOOLEAN type, namely, as conditions in if statements, exit statements, select statements, assert statements, and while clauses.

Test Objectives and Design Guidelines

1. Check that the arguments to BOOLEAN attributes (SUCC, PRED, POS, IMAGE) can only be BOOLEAN values. (Since BOOLEAN is predefined and therefore built into an implementation, it is possible that the appropriate type checks will not be made for these attributes.)

3.5.5 Attributes of Discrete Types and Subtypes

Semantic Ramifications

Note that SUCC, PRED, and POS are applicable to integer types as well as enumeration types.

The forms SUCC(X) and PRED(X) cannot be used to invoke the predefined successor and predecessor functions (LRM 4.1.2) unless a renaming declaration has been used (see 8.5). Otherwise, a discrete type or subtype name must precede the attribute name.

Compile-time Constraints

1. The argument to SUCC, PRED, or POS must be of a discrete type.

2. The base type of T and the base type of the argument to T'SUCC, T'PRED, and T'POS must be the same.

Exceptions

1. Evaluation of T'SUCC raises CONSTRAINT_ERROR if its argument equals the last value in the base type of T.

2. Evaluation of T'PRED raises CONSTRAINT_ERROR if its argument equals the first value in the base type of T.

3. Evaluation of T'VAL(I), where I is an integer value, raises CONSTRAINT_ERROR if I is not in the range B'POS(B'FIRST)..B'POS(B'LAST), where B is the base type of T.

4. Evaluation of T'VALUE(S), where S is a string value, raises DATA_ERROR if there exists no X such that T'IMAGE(X) is identical to S, except for possible differences in case.

5. Evaluation of T'VALUE(S) raises CONSTRAINT_ERROR if its value lies outside the range T'FIRST..T'LAST.

Test Objectives and Design Guidelines

1. Check that the argument to SUCC, PRED, and POS must be of a discrete type. (Use strings of length one among other cases.)

2. Check that any value of the base type of T can be used as an argument to T'SUCC, T'PRED, and T'POS (see 3.5.1.T/5).

3. Check that:

   - VALUE and IMAGE give correct results.

     Implementation Guideline: Use mixed case string arguments to VALUE and determine whether IMAGE maintains the case of the enumeration identifier given in the program or makes a systematic change of case.

   - PRED and SUCC give correct results when no exceptions should be raised (see also T/6).

   - VAL and POS produce correct results when no exceptions are raised.

     Implementation Guideline: Be sure to check POS and VAL for subtypes, e.g., WEEKDAY'POS(MON).

     Implementation Guideline: Check T'VAL(T'POS(X)) = X for every X in T, including subtypes of T.

4. Check that CONSTRAINT_ERROR is not raised when the enumeration argument to SUCC, PRED, POS, VAL, IMAGE, and VALUE does not satisfy T's subtype constraints.

   Implementation Guideline: Use both a type and subtype of an enumeration type in these tests.

5. Check that T'SUCC and T'PRED raise CONSTRAINT_ERROR when their arguments equal B'LAST and B'FIRST, respectively, where B is the base type of T.

   Implementation Guideline: Use both types and subtypes for T such that if S is a subtype of T, S'FIRST /= T'FIRST and S'LAST /= T'LAST.

6. Check that the argument to T'SUCC, T'PRED, and T'POS must have the base type of T.

   Implementation Guideline: Try at least one test using SUCC with a non-discrete type, e.g., a fixed point type with a delta of 1.0.

7. Check that the FIRST, LAST, PRED, SUCC, POS, VAL, IMAGE, and VALUE attributes for the predefined character type CHARACTER yield the appropriate values. Check that CONSTRAINT_ERROR is raised under appropriate circumstances. (Note: 2.5.T/7 checks that all standard CHARACTER literals have the correct physial representation.)

   Implementation Guideline: Since the physical and logical representation for CHARACTER values is essentially the same, PRED, SUCC, and POS need be checked for only a few values. IMAGE and VALUE, however, should be checked for all CHARACTER values, including the alternative names for characters, e.g., DOLLAR, LC_A, etc. (However, see Gaps, below.)

8. Check that the FIRST, LAST, PRED, SUCC, POS, VAL, IMAGE, and VALUE

attributes for the predefined BOOLEAN type yield the appropriate values. Check that CONSTRAINT_ERROR is raised under appropriate conditions.

Implementation Guideline: In checking these attributes, use expressions such as A < B as arguments, e.g., BOOLEAN'POS(A < B), as well as simple literals and variables.

Gaps

1. It is unclear what CHARACTER'IMAGE(LC_A) is to return for an implementation. Presumably, CHARACTER'IMAGE(LC_A) = CHARACTER'IMAGE('a') if the host computer supports lower case characters, but the actual value returned would seem to depend on the character set supported by the target computer. Hence, it appears the value is implementation-dependent, i.e., either "LC_A" or "'a'" (see LRM Appendix A).

3.5.6 Real Types

Semantic Ramifications

No specific tests or comments can be made on this section since the details of fixed and floating points types differ. The syntax given in this section is not sufficient for testing in itself.

The concept of "model" floating and fixed point numbers is used to specify the minimum accuracy and range of values an implementation must provide for values of a given real type. An implementation is free to use a more accurate representation than specified by a type definition, as discussed below. However, some machines provide overlength accumulators, so computations with "single precision" operands produce results with more bits of precision than is implied by the stored representation of values. In effect, different representations are being provided for the same real type -- a stored representation and a representation used for results of computations. If an optimizing compiler keeps results in registers and these registers give more bits of precision than the stored values, then a comparison such as A*B > C may be true or false depending on whether C is the result of another computation, and hence, is in a register, or is a stored value. Such uncertainties in computing with real values are reflected in the properties of Ada's model for real numbers, and hence, is permitted. But an implementation that takes advantage of the extra accuracy provided by a machine may surprise its users, so care should be taken.

Certain fixed and floating point subtypes are defined to yield a smaller set of model numbers than their base type has. The model permits such subtypes to be represented with a shorter representation than the base type, e.g., a subtype might be defined that permits a single precision representation of what would otherwise require a double precision representation. We call this use of a different representations for subtypes, "subtype optimization". It is not recommended as an implementation technique since it is difficult to maintain the non-precision-related properties of the base type (e.g., properties dependent on 'FIRST and 'LAST) while using a

different representation for a subtype. Tests have been devised to see if an implementation is performing subtype optimization, and if so, if it is performing it correctly.

An algorithm can easily use more than the guaranteed properties of a real type and hence violate the portability rules. A diagnostic compiler could issue warnings about such transgressions but in many cases where portability is not a requirement the issuing of a warning is not useful. In fact, the programmer can achieve portability by one of two means, firstly by using only the guaranteed Ada properties of real types or secondly by using machine parameters of the underlying hardware types. The Ada method is crude but simple whereas the machine method can give a closer match to the actual hardware at the expense of some additional complexity.

The test programs on real types do not violate the portability rules except where stated.

3.5.7 Floating Point Types

Semantic Ramifications

A floating point type definition guarantees minimal properties. These are based upon the exact handling of model numbers. An implementer must characterize the basic hardware types by means of the predefined attributes (see 3.5.8.S). When defining a floating point type F, the floating point constraint in the type definition is then used to select the appropriate hardware type by ensuring that F'BASE'DIGITS > F'DIGITS. Note that one would usually choose the least accurate type which satisfies this inequality. Since the properties of F are guaranteed from those of F'BASE, it is only necessary for the implementer to ensure that the hardware types are handled correctly. For instance, if F'DIGITS=5 and F'BASE'DIGITS=8, then F's values can be handled as if they had 8 digits of precision. This strategy is simple and sufficient although it is not necessary since the user only asked for 5 digits of precision.

Note that the values of L and R given in the range constraint can lie outside the range of model numbers for the type, since a floating point type declaration is implicitly a derived type declaration. The implementer, however, is only required to choose an underlying hardware type based on the value of D. For example, given

```
type F is digits D range L .. R;
```

suppose FLOAT'DIGITS > D but FLOAT'LAST < R. Suppose LONG_FLOAT'LAST > R. An implementation is permitted to choose F'BASE = FLOAT, so the implied declaration of F is

```
type F is new FLOAT digits D range L .. R;
```

and CONSTRAINT_ERROR will be raised since FLOAT'LAST < R (see 3.3). But an implementation could also choose F'BASE = LONG_FLOAT, in which case,

CONSTRAINT_ERROR will not be raised. The language permits, but does not require the implementation to take into account the values of L and R as well as D in choosing a base type. If the values of L and R are taken into consideration, then different base types might be choosen for the same digits value (i.e., for the same range of model numbers (see G/2)). In any event, if the chosen type will cause CONSTRAINT_ERROR to be raised when the declaration is elaborated, the compiler is permitted to issue a warning at compile time (see 10.6). Note that if the range constraint is not given, then F'LAST = F'BASE'LAST, which is not, in general, a model number (and simiiarly, for F'FIRST).

The Ada model of floating point is based upon the work of W S Brown, the major changes being to fix the radix at 2 and to require an exponent range which depends upon the mantissa length. With these two points taken into account, the best detailed commentary on the concept of model numbers is the paper by W S Brown "A simple but realistic model of floating-point computation" Computer Science Technical Report No 83, May 1980. Bell Laboratories, NJ 07974. Another minor change is that division is a supported operation, which is not true for a few computers such as the Cray 1 and Interdata 8/32.

Literal values are in general obtained from literal expressions (see 4.10). If the values are exactly model numbers, they must be converted exactly. If they are not, they must lie in the appropriate model interval (see 4.6). There is no requirement that literal values which are not model numbers be converted consistently. For instance, consider:

```
X := 0.1;
-- some calculation not changing X
if X = 0.1 then
```

then the condition is not necessarily true. The reason is that on some machines computations are performed more accurately than results can be stored in main memory (overlength accumulator). Hence the value of X depends upon whether a register value or a stored value is used - which in turn, with an optimizing compiler, would depend upon the intermediate computation above. A compiler need not (and should not) store constants with only minimal accuracy if the machine can do more at no additional cost. Hence in the above example on a machine with an overlength accumulator where constants can be loaded overlength, the value 0.1 can be different from X when X is stored in main memory.

If the hardware of the machine does not support the Ada model for underflow (i.e., NUMERIC_ERROR is not raised if any non-zero value just inside the range -F'SMALL .. F'SMALL is produced) and if the machine does not provide a method of suppressing underflow, then a handler may be necessary for underflow which returns control to the point of an underflow interrupt. It may be possible by software to provide either gradual underflow or zero when underflow occurs, since the model permits either option. Gradual underflow has a few advantages (it is part of the IEEE Standard, for instance; see SIGNUM Newsletter, October 1979), but if this increases execution time for every underflow encountered, then zero is probably better. For overflow, the

language permits, but does not require, NUMERIC_ERROR to be raised. What should be done when overflow occurs depends on the target computer, since Ada does not specify any required behavior. For example, an implementation may have the choice of raising NUMERIC_ERROR or continuing with an infinity value when the implemented range is exceeded. This choice could be left to the user by means of a pragma. In general, raising the exception is to be preferred on the grounds that programming with infinity values cannot be done in a portable manner (such values are, of course, outside the Brown model).

If an implementation wishes to provide arithmetic with "infinite" values, this must be done with care. The attributes 'FIRST and 'LAST are always defined for a floating point type and hence with the IEEE projective mode there is a problem since there is only one unsigned infinity. One could set 'FIRST to the largest negative value and 'LAST to infinity provided the inequalities gave the implemented range as being 'FIRST to 'LAST. The use of interrupt routines may be needed to give these inequalities. This problem arises because 'FIRST and 'LAST need not be model numbers but yet have properties which involve the relational operators:

```
for all X in T: X >= T'FIRST
for all X in T: X <= T'LAST
```

For limitations on the accuracy of the floating point types that an implementation provides, see the definition of SYSTEM.MAX_DIGITS under Gaps.

Since a type declaration using a floating point type constraint introduces a new floating point type derived from a predefined floating point type, the usual rules for derived types apply (see 3.4). In particular, the predefined operations for floating point types are implicitly declared in the declarative part containing the type declaration. These operators are the relational and membership operators, "+", "-", "*", "/", "**", and ABS (see Appendix C and 4.5.2).

Note that in a type definition of the form <u>digits</u> D <u>range</u> L .. R, L and R need not be expressions of the same floating point type, nor need they be of a floating point type -- they can be of any fixed point type as well.

Note that if an implementation supports SHORT_FLOAT and/or LONG_FLOAT, SHORT_FLOAT'DIGITS < FLOAT'DIGITS < LONG_FLOAT'DIGITS must hold.

<u>Compile-time Constraints</u>

1. In a floating point constraint, the expression following <u>digits</u> must be a static expression of an integer type and its value must be less than or equal to SYSTEM.MAX_DIGITS (see Gaps) and greater than zero.

2. In a subtype indication of the form T <u>digits</u> D <u>range</u> L .. R, the base types of T, L, and R must be the same.

3. A floating point constraint can be specified in a subtype indication only if the type_mark denotes a floating point type.

4. If a range_constraint is provided in a floating point constraint used in a type_definition, the bounds of the range_constraint must be specified as static expressions of real types.

5. A null range cannot be specified in a type definition for a floating point type (see Gaps).

Exceptions

1. CONSTRAINT_ERROR is raised (see 3.3) for a floating point constraint used in a subtype indication of the form T digits D [range_constraint] if D is greater than T'DIGITS or less than zero. (This check cannot be suppressed since the definition of RANGE_CHECK does not apply in this case; see Gaps.)

2. CONSTRAINT_ERROR is raised (see 3.3 and 3.5.0) if in a subtype indication of the form T digits D range L .. R, (L in T) and (R in T) is FALSE. (This check can be suppressed by applying RANGE_CHECK to T.)

3. CONSTRAINT_ERROR is raised in a type definition of the form digits D range L .. R if the value of L or R does not lie in the range of the selected predefined floating pint type. (This check can be suppressed by applying RANGE_CHECK to the selected predefined floating point type.)

Test Objectives and Design Guidelines

Some tests need to be repeated for each floating point type. This is done by a test template containing a type definition for digits N, N in 5 .. 30. The 26 tests are denoted by the final letters A .. Z.

1. Check that

   a. the expression after digits must be an integer type.

   b. the expression after digits must not be negative or zero.

   c. the expression after digits must be static.

   d. the expression after digits must have a value <= SYSTEM.MAX_DIGITS.

   e. the expressions in a floating point range_constraint must not be of an integer type.

   f. a null range cannot be specified in a floating point type_definition.

   g. FLOAT is provided as a predefined type.

2. Check that SHORT_FLOAT'DIGITS < FLOAT'DIGITS and FLOAT'DIGITS < LONG_FLOAT'DIGITS.

3. Check that FIRST and LAST can be assigned without raising CONSTRAINT_ERROR and that FIRST <= LAST for the predefined types.

4. Check that L and R in a type definition's range constraint can be fixed point or different real types.

5. For each of digits 5..30, check that

   a. values corresponding to LARGE, -LARGE, SMALL, -SMALL, EPSILON, and 1.0+EPSILON can be assigned and used in equality relations.

   b. negative powers of 2.0 down to 2.0**(-30) are represented exactly.

   c. that a literal value which is not a model number lies in the appropriate model interval (see 3.5.6).

6. Check that

   a. two types with the same textual declaration are distinct,

   b. two types derived from a single type are distinct,

   c. subtypes of distinct types are distinct.

Gaps

1. The names of the predefined types other than SHORT_FLOAT, FLOAT, and LONG_FLOAT are not specified in the LRM.

2. The language does not require that the choice of hardware type is consistent, i.e. the choice is requied only to be a function of D. The user would expect consistency and any inconsistency should be clearly stated in the compiler documentation. An inconsistency could arise as a consequence of a compiler update, but such an update should presumably require complete recompilation since otherwise, units compiled after the update will not be compatible with previously compiled units. The choice could be influenced by a pragma.

3. The manual does not make it clear if a null range in a floating point type definition is permitted. In section 3.5, under scalar types, null ranges are permitted. Null ranges are not permitted in integer type definitions or in enumeration types. We have assumed a null range is forbidden.

4. The maximum number of digits which an implementation provides should be available as a constant in the standard environment as SYSTEM. We have assumed the presence of SYSTEM.MAX_DIGITS.

3.5.8 Attributes of Floating Point Types

Semantic Ramifications

The predefined attributes which do not start with MACHINE (see 13.7.1) are dependent on the model and hence have precise properties. Since the precision can only be specified in decimal digits, the attribute 'MANTISSA

cannot have certain values. In practice 'DIGITS is likely to be in the range 5..30, and hence it is possible to tabulate all possible values. Note that for 'DIGITS = D, 'LARGE is approximately 10**(4*D) to 10**(4*D + 1.2).

DIGITS	MANTISSA	EMAX	SMALL	LARGE	EPSILON
5	17	68	1.694E-21	2.952E20	1.526E-5
6	20	80	4.136E-25	1.209E24	1.907E-6
7	24	96	6.311E-30	7.923E28	1.192E-7
8	27	108	1.541E-33	3.245E32	1.490E-8
9	30	120	3.762E-37	1.329E36	1.863E-9
10	34	136	5.740E-42	8.711E40	1.164E-10
11	37	148	1.401E-45	3.568E44	1.455E-11
12	40	160	3.421E-49	1.462E48	1.819E-12
13	44	176	5.220E-54	9.578E52	1.134E-13
14	47	188	1.274E-57	3.923E56	1.421E-14
15	50	200	3.112E-61	1.607E60	1.776E-15
16	54	216	4.748E-66	1.053E65	1.110E-16
17	57	228	1.159E-69	4.314E68	1.388E-17
18	60	240	2.830E-73	1.767E72	1.735E-18
19	64	256	4.318E-78	1.158E71	1.084E-19
20	67	268	1.054E-81	4.743E80	1.355E-20
21	70	280	2.574E-85	1.943E84	1.694E-21
22	74	296	3.927E-90	1.273E89	1.059E-22
23	77	308	9.588E-94	5.215E92	1.323E-23
24	80	320	2.341E-97	2.136E96	1.654E-24
25	84	336	3.572E-102	1.400E101	1.034E-25
26	87	348	8.720E-106	5.734E104	1.292E-26
27	90	360	2.129E-109	2.349E108	1.616E-27
28	94	376	3.248E-114	1.539E113	1.010E-28
29	97	388	7.931E-118	6.304E116	1.262E-29
30	100	400	1.936E-121	2.582E120	1.578E-30

The values of SMALL, LARGE and EPSILON are rounded to four figures. Hence the rounding up of LARGE in the above table could make the value greater than the exact value of LARGE as computed by

```
2.0**F'EMAX * (1.0 - 2.0**(-F'MANTISSA))
```

Note that the requirement for the exponent range may restrict the value of MANTISSA. For instance, on the Honeywell 6000 in double precision, the exponent range is only to 2.0**127 restricting the value of MANTISSA to less than 32. For the decimal equivalent, one requires that MANTISSA=30 and DIGITS=9 which is approximately half the actual precision of the machine. In almost all other cases, the actual mantissa length restricts the value of MANTISSA and hence the model guarantees a rather smaller exponent range than is actually provided. (The value 4*B as 'EMAX was chosen because some minimal

exponent range had to be guaranteed by a model that only used the value of digits to determine a model range, and 4*B was thought to be a reasonable compromise between user needs and what most machines provide.)

For a detailed discussion of the problems in determining the correct attribute settings for floating point, the reader should consult: "Environmental parameters and basic functions for floating point computation" Computer Science Technical Report No 72, April 1980, Bell Laboratories, NJ 07940, by W S Brown and S I Feldman. The problems with Ada are somewhat less severe than those described in the paper since in Ada the radix is 2 and the definition of floating point types in terms of decimal digits means that the choice is more coarse. Linking the exponent range to the mantissa length causes slight additional problems.

Test Objectives and Design Guidelines

1. Check that 'DIGITS, 'MANTISSA, and 'EMAX are of type universal integer and 'SMALL, 'LAREGE, and 'EPSILON are of type universal real.

2. Check values of attributes for all digits N, N=5..30: namely the

   a. value of 'DIGITS, and value <= 'BASE'DIGITS,

   b. value of 'MANTISSA, and value <= 'BASE'MANTISSA,

   c. value of 'SMALL, 'LARGE and 'EPSILON.

3. Check unnormalized based numbers which are model numbers, and spot check that some non-model number constants are bounded by model numbers.

3.5.9 Fixed Point Types

Semantic Ramifications

Fixed point types correspond to the use of integer hardware to represent non-integral values. The implementation can choose the ACTUAL_DELTA unless it is set by a representation specification. Special checks that are appropriate to a specified ACTUAL_DELTA are handled under Chapter 13. Without such a specification, an implementation will probably choose a power of 2.0 (positive or negative) as the value of ACTUAL_DELTA. This choice considerably eases conversions between fixed point types and other numeric types. Note that fixed point type conversion is very common because of the need to rescale after multiplication and division.

Comparisons between fixed point values must not raise NUMERIC_ERROR. If a machine requires the use of subtraction to compare values, then either long sequences may be needed (to test sign first) or a spare bit at the top of the word can be left free so that subtraction is safe. Leaving a spare bit at the top is awkward since it will be necessary to check that computed values do not exceed the range 'FIRST .. 'LAST, which does not include the spare bit (see Gaps, however).

An implementation would typically choose one or two words for a fixed point data type. Larger lengths handled by subroutine would be possible but the overheads are such that a programmer would probably prefer floating point because of the automatic scaling. For limitations on the implemented accuracy, see the definition of MIN_DELTA under 'Gaps'.

Compile-time Constraints

1. A fixed point constraint used in a type_definition must have the following properties, where DELTA is the value specified after delta and the values of the lower and upper bounds in the range constraint are called L and R, respectively:

   - DELTA must be a static expression of a real type greater than SYSTEM.MIN_DELTA (see Gaps);
   - a range constraint must be provided:
   - L and R must be static expressions of real types (not necessarily the same real types);
   - a null range cannot be specified (see Gaps);
   - DELTA must be less than the maximum of ABS(L) and ABS(R);
   - If I is the smallest integer larger than or equal to ln(MAX(ABS(L),ABS(R)))/ln(2.0), and J is the smallest integer larger than or equal to ln(DELTA)/ln(2.0), and K is the smallest integer larger than or equal to ln(MIN_DELTA)/ln(2.0), then the fixed point type will be supported if I-J <= -K. Note that I-J is the attribute 'BITS (assuming a power of two for ACTUAL_DELTA), and hence the condition is that the type definition should not require more bits than the maximum implied by MIN_DELTA.

2. In a subtype indication of the form T delta D range L .. R, the base types of T, L, and R must all be the same.

3. A fixed point constraint can be specified in a subtype indication only if the type_mark denotes a fixed point type.

Exceptions

1. CONSTRAINT_ERROR is raised (see 3.3) for a fixed point constraint used in a subtype indication of the form T delta D [range_constraint] if D is less than T'DELTA (see Gaps). (This check cannot be suppressed since the definition of RANGE_CHECK does not apply in this case.)

2. CONSTRAINT_ERROR is raised (see 3.3 and 3.5.0) if in a subtype indication of the form T delta D range L .. R, the expression (L in T) and (R in T) is FALSE. (This check can be suppressed by applying RANGE_CHECK to T.)

Test Objectives and Design Guidelines

In order to check properties of fixed point types without invoking generics (and with the ability to make specific tests for special cases), some tests are repeated for a range of type definitions.

1. Check that

   a. DELTA cannot be integer.

   b. L or R cannot be integer.

   c. DELTA is static.

   d. DELTA<0.0 fails.

   e. a large range with a small delta is forbidden.

   f. L and R must be static.

   g. a range constraint cannot be omitted in a type definition.

2. Check that:

   a. a type with only 3 model numbers is handled correctly.

   b. a large range and large delta is permitted and handled correctly.

   c. a small range with a very small delta is permitted and handled correctly.

   d. DELTA, L, and R can be floating point.

   e. L and R can be of different real types.

   f. the values of L and R need not be model numbers, and both ABS(L) and ABS(R) must either be less than the largest model number or the difference between ABS(L) (or ABS (R)) and the largest model number must be less than the specified value of DELTA.

3. For a number of fixed point types, check that

   a. model numbers are stored exactly and

   b. if ACTUAL_DELTA is a power of 2, check that powers of 2 are stored exactly,

   c. the representation of a literal value which is not a model number lies in the appropriate model interval.

Gaps

1. The manual does not make it clear whether a null range in a type definition is permitted. In section 3.5, under scalar types, null ranges are permitted. Null ranges are not permitted in integer type definitions or in enumeration types. We have assumed null ranges are not permitted.

2. The maximum accuracy of an implemented fixed point type should be provided in the standard environment. This could be the smallest value MIN_DELTA such that:

   <u>type</u> F <u>is</u> <u>delta</u> MIN_DELTA <u>range</u> -1.0+MIN_DELTA .. 1.0-MIN_DELTA;

   is implemented. Since this value is needed for the test programs and is also needed to specify restrictions on an implementation, its absence from the required standard environment is noted as a gap. We have assumed the value SYSTEM.MIN_DELTA is available.

3. In a subtype indication of the form T fixed_point_constraint, the check that T'DELTA is less than or equal to the delta specified in the constraint can always be performed at compile time. However, checks of the compatibility of a constraint in a subtype indication are defined to raise CONSTRAINT_ERROR (see 3.3), and so, although a compiler can issue a warning if the relation between the delta values will raise CONSTRAINT_ERROR, it cannot reject the program as illegal. This may be an oversight.

4. The definition that a value of a fixed point type satisfies a fixed point constraint if it satisfies any included range constraint is inconsistent with the definition of membership operators in 4.5.2, where the LRM says the membership operations test "whether (a value) satisfies any constraint imposed by a subtype indication. ... A test for an accuracy constraint always yields the result TRUE." Presumably the statement in 4.5.2 was intended to apply only to accuracy constraints that do not specify an explicit range_constraint (see 4.5.2.G).

5. The method by which fixed point operators are introduced by a fixed point type definition is not stated in the LRM, since fixed point types are not defined in terms of derived types. By analogy with the other numeric types, however, one would expect that the operators applicable to a specified fixed point type are introduced in the declarative part where the type is declared, and hence can be named as selected components of the declarative part and are only available for the type if directly visible (see 3.4.S).

6. Related to G/5 is the fact that the base type of a fixed point type is not defined in the LRM. Presumably the base type reflects the actual representation chosen for the fixed point type, and hence, for a fixed point type F declared with range bounds L and R, F'BASE'FIRST does not necessarily equal F'FIRST, i.e., L. With an appropriate definition of a base type, given variables A and B of type F, A + B would presumably not raise CONSTRAINT_ERROR if A + B was outside F'FIRST .. F'LAST, i.e., "+" would be defined in terms of the base type, not the defined (sub)type.

3.5.10 Attributes of Fixed Point Types

Semantic Ramifications

The values of the attributes cannot be listed since DELTA is potentially any real value. Also, ACTUAL_DELTA can be set to any real value by a representation specification if the implementation supports this. The simplest conceivable implementation of fixed point would be to allow only one size (implying an ACTUAL_DELTA dependent on the word length), and not permit any representation specification which conflicts with this. However, since fixed point is often needed for packing real values and for processing digital input, representation specifications are likely to be a requirement.

Test Objectives and Design Guidelines

Since the likely default value for ACTUAL_DELTA is a power of 2, several of the tests are written with this assumption. As a consequence, these tests may be be as searching if the implementation does not choose this default. Even when a power of 2 is taken, an implementation is likely to choose a value which implies taking a whole number of machine words.

1. Check that

   a. 'BITS is universal integer,

   b. 'DELTA, 'ACTUAL_DELTA and 'LARGE are universal real.

2. For a variety of fixed point types, check

   a. the formula for 'LARGE,

   b. 'DELTA >= 'ACTUAL_DELTA,

   c. L >= -F'LARGE - F'DELTA in a fixed point range constraint, and R <= F'LARGE + F'DELTA.

3.6 Array Types

3.6.1 Index Constraints and Discrete Ranges

This section discusses discrete ranges and index constraints separately. Since index constraints use discrete ranges, discrete ranges are discussed first.

3.6.1.a Discrete Ranges

Semantic Ramifications

A discrete_range can be used to specify:

- a choice in a case statement, variant part, or aggregate;
- a slice;
- an iteration_clause in a loop;
- an index in an array_type_definition;
- an index_constraint associated with an array type; and
- a family of entries.

In iteration_clauses and array_type_definitions, a discrete range defines the type of the loop parameter and array index, respectively, as well as constraining the set of acceptable values. In the other contexts, a compiler must determine whether the discrete_range has, or can be given, a type consistent with the type required by the context.

We will first discuss discrete_ranges in which a type_mark appears explicitly, and then the form in which just the range is given.

Explicitly Typed Discrete Ranges

The type associated with a discrete_range determines what values can be specified for a range. If the type_mark in a discrete_range is a subtype name, ST, ST's base type is the type associated with the discrete_range. For example, given

```
type DAY is (MON, TUE, WED, THU, FRI, SAT, SUN);
subtype WEEKDAY is DAY range MON .. FRI;
subtype MIDWEEK is WEEKDAY range TUE .. THU;
```

the discrete range:

```
MIDWEEK range WED .. THU
```

has the type DAY. LRM 3.5.0 states that only values of MIDWEEK's base type, DAY, can appear in this discrete_range. In addition, for a discrete_range of the form:

```
ST range L .. R
```

if R is greater than or equal to L (i.e., the range is not null), the values of L and R must satisfy the range constraints of ST. Hence,

```
MIDWEEK range WED .. FRI
```

would raise the exception CONSTRAINT_ERROR, since FRI is not in MIDWEEK's set of values.

However, when the discrete range is null (i.e., R = 'PRED(L), it is still required that the value of L satisfy the range constraint of ST, but the value of R need only be a value of ST's base type. Hence,

```
MIDWEEK range TUE .. MON
```

is permitted and raises no exceptions.

A discrete_range of the form ST can be considered an abbreviation for:

```
ST range ST'FIRST .. ST'LAST
```

The array attribute 'RANGE (see 3.6.2) is a form of subtype name. All the remarks in this section applying to type_marks apply to the attribute 'RANGE as well.

Discrete Ranges Without Type Marks

When a discrete_range does not contain a type_mark, the type of the discrete_range is determined by the type of the expressions used to define the lower and upper bounds and from the context in which the discrete_range appears (see below for a definition of the contexts to be considered). For example, the following discrete_ranges are equivalent to ranges preceded by "DAY range", if these literals are all of type DAY:

```
MON .. FRI
SUN .. SAT
TUE .. SAT
```

When a discrete_range of the form L .. R contains only integer literal expressions, the type of L and R must be determined from the context, except in two cases:

- when the discrete range is used in an iteration clause, and
- when the discrete range is used as an index in an array type definition.

In these cases, L and R are assumed to be of type INTEGER. This means, for example, that in a loop statement

```
for I in 1 .. 10 loop
```

I cannot have the type SHORT_INTEGER, even if an implementation supports this type and SHORT_INTEGER'LAST > 10. Moreover, if 1_000_000 is greater than INTEGER'LAST, then 1 .. 1_000_000 raises CONSTRAINT_ERROR in these contexts, since it is equivalent to

```
for I in INTEGER range 1 .. 1_000_000 loop
```

and 1_000_000 exceeds INTEGER'LAST (see 3.5.4).

If L and R are not both integer literal expressions, context is used to determine their type. If the context is not sufficient, the program is illegal. In the case of iteration clauses and array type definitions, the only usable contextual requirement is that L and R must have the same type. In other constructs, additional restrictions can (and must) be used to resolve any uncertainty in the type of L and R:

- in case statements, the type of each choice must match the type of the case expression (which cannot be overloaded);
- in variant parts, the type of a choice is determined by the type of the discriminant (since the discriminant cannot be ambiguous in its type);
- in an array aggregate, all the choices must have the same type and all the components must have the same type, although the type of the components need not, of course, be the same as the type of the choices (see 3.6.2 for futher discussion);
- in a slice, the type of the discrete range is determined by the type of the array being sliced. (Note that when the result of a function is sliced, the type of the array returned by the function is contextually determined.)
- in an index_constraint, the type of the discrete range is determined by the type of the array being constrained and the type of the bounds for each dimension.

The above restrictions must be sufficient to uniquely determine the type of L and R; otherwise the program is illegal. Examples for each context are given in the relevant sections of this Guide.

Overloading Resolution

Both the lower and upper bound of a range must have the same type. This restriction must be taken into account when resolving the type of overloaded literals or expressions used for bounds. For example, if the enumeration type

```
type SOL is (SUN, MERCURY, VENUS, EARTH, MARS, ...);
```

is in the same scope as DAY, then SUN .. TUE is a discrete_range of type DAY and SUN .. VENUS is a discrete_range of type SOL.

Although the discrete_range must be a discrete type, this restriction cannot be used to resolve an overloading ambiguity. For example, if F and G are parameterless functions that return either INTEGER or FLOAT values, then

```
for I in F() .. G() loop ... end loop;
```

is ambiguous (and hence illegal) even though only functions returning discrete, i.e., INTEGER, values are legal here. However,

```
for I in 2 .. F loop ... end loop;
```

would be unambiguous, since 2 cannot be of type FLOAT. However, if we introduce a new type and function definition:

```
type T is new INTEGER;
function F return T;
```

then

```
for I in 2 .. F loop
```

is ambiguous, since 2 can either be of type T or type INTEGER.

Overloading resolution is discussed further in 6.6.

Compile-time Constraints

1. A discrete_range must have a discrete type.

2. The bounds of a discrete_range must have the same type.

3. For a discrete_range of the form ST range L .. R, L and R must both be values of type ST'BASE. (Note: if L or R does not satisfy ST's range constraints, the program is not illegal. Instead an exception is raised at run time if the discrete_range is evaluated. See LRM 10.6).

Exceptions

For a null discrete range, we suspect (see Gap G/3) that the intent is for CONSTRAINT_ERROR to be raised only when a value of the range constraint is outside the base type's range. This is only possible for integer literal values (see 3.5.4).

1. For a discrete non-null range of the form L .. R, where L and R are of type ST, or the form ST range L .. R, if

   (L not in ST) or (R not in ST)

   then CONSTRAINT_ERROR is raised (see Gaps). (This check is suppressed by applying RANGE_CHECK to ST.)

Test Objectives and Design Guidelines

The contextually determined validity of discrete ranges is checked in each context in which discrete ranges can be used. We only check here for context-independent discrete range validity.

1. Check that the type_mark (if present) and both bounds must be discrete types.

   Implementation Guideline: Check all illegal combinations of the following:

- type_mark non-discrete/discrete/absent
- lower bound non-discrete/discrete
- upper bound non-discrete/discrete
- bounds are literals/non-literals

All combinations should be tried for discrete_ranges in both loops and array type definitions. At least two illegal combinations should be tried for discrete ranges in case statement choices, variant part choices, array aggregates, slices, and as index_constraints in an object declaration and a type declaration. Different forms of illegal discrete ranges should be used in each of these checks.

2. Check that the upper and lower bound cannot have different discrete types. Use the form with and without a type_mark.

   Implementation Guideline: Try at least one example of a discrete range in each of the contexts permitting discrete ranges.

3. Check that when a type_mark is present and both bounds are of the same type, the type of the bounds must be the type associated with the type_mark.

   Implementation Guideline: Use both a subtype and type name for the type_mark, and use non-null ranges; limit this test to loops and array type definitions.

4. Check that for the form ST range L .. R, CONSTRAINT_ERROR is raised if the range is non-null and L or R are outside ST'FIRST .. ST'LAST.

   Check that no exception is raised if L = ST'FIRST and R = ST'PRED(L) >= ST'BASE'FIRST.

   Check that CONSTRAINT_ERROR is raised if R < L and L not in ST or R /= ST'PRED(L).

   Implementation Guideline: Use both static and non-static out of range expressions for L and R, but write separate tests for the static and non-static cases. For null ranges, use both enumeration and INTEGER subtypes in loops, slices, and membership operations. For non-null ranges, check all contexts in which a discrete range can appear. Use A'RANGE for ST.

5. Using non-null discrete ranges of the form L .. R in loop and array type definitions, check that CONSTRAINT_ERROR is raised if L and R are integer literal exressions and at least one of them is not in the range of INTEGER values.

   Check that CONSTRAINT_ERROR is raised if for the form L .. R, R < L and R /= L-1.

In a separate test, check that neither the loop parameter nor the array index is assumed to be of the type SHORT_INTEGER if L and R are both in the range of SHORT_INTEGER values.

Implementation Guideline: Use the loop parameter as a subscript for an array having a SHORT_INTEGER index type.

50. Check that functions overloaded to return a discrete and non-discrete result are considered ambiguous if used as discrete range bounds in a loop or array type definition where the type of the discrete range is not given explicitly and neither bound is uniquely a discrete type.

    Check that if one bound is a literal or expression of a discrete type, the overloading for the other bound is successfully resolved.

51. Check that functions overloaded to return an INTEGER type and some other discrete type (including a derived integer type) are considered ambiguous if used as discrete range bounds in a loop or array type definition where the type of the discrete range is not given explicitly.

Gaps

1. When a discrete_range is used to slice the result of an overloaded function, it is unclear to what extent the bounds or type of the discrete range are used to resolve the overloading.

2. The LRM states that the exception condition CONSTRAINT_ERROR is raised for all null discrete ranges such that the upper bound is less than the predecessor of the lower bound. However, it was probably the intent (see also next Gap) that this exception only be raised for discrete ranges used as null index ranges in index constraints, slices, and entry_declarations, but not ranges used in choices or iteration clauses.

3. The statement "The exception CONSTRAINT_ERROR is raised for any incompatible discrete range or if the upper bound of a discrete range is less than the predecessor of the lower bound" is unsatisfactory in several respects:

    - It does not specify when CONSTRAINT_ERROR is raised for non-null ranges if the range_constraint in the discrete_range is incompatible with the type_mark of the discrete_range. We assume that CONSTRAINT_ERROR is raised if the range_constraint is incompatible with the type_mark of the discrete range, where incompatible is defined in 3.5.0. Note that incompatability of the discrete range's range_constraint is different from incompatibility of the discrete_range's use to specify an index range. For use as an index range, incompatability is defined in terms of the type_mark for an array index, not the type_mark used in the discrete range itself, e.g.,

```
subtype LOW is INTEGER range 1..7;
subtype MED is INTEGER range 5..10;
```

```
type T is array (LOW range <>) of INTEGER;
X: T(MED range 1..7);    -- CONSTRAINT_ERROR
```

CONSTRAINT_ERROR should be raised here because even though 1..7 is compatible with T's index type, it is incompatible with the range associated with MED, i.e., CONSTRAINT_ERROR should be raised because MED range 1..7 is an invalid discrete_range. The currrent wording does not clearly cover this case.

- The definition of when a null discrete range raises CONSTRAINT_ERROR applies to all discrete_ranges, not just those used in entry declarations, index constraints, and slices. Consequently, a loop of the form for I in 1 .. N loop raises CONSTRAINT_ERROR if N is not equal to zero. Although this was almost certainly not intended, there is no ambiguity in the current wording, and so we do not reflect the intended restriction in this version of the Guide. Implementers may (at their own risk, of course) decide not to raise CONSTRAINT_ERROR in such loop statements on the assumption that the final version of the Standard will correct this error.

- The rule does not specify that for a null range of the form ST range L .. R, L must be a value in ST's range. Hence, if ST is declared as INTEGER range 1..10, ST range -49..-50 does not raise CONSTRAINT_ERROR. Whether or not CONSTRAINT_ERROR is raised for an index range depends (according to the current wording) only on the characteristics of the index type:

```
subtype ST is INTEGER range 1..10;
type A_1 is array (INTEGER range <>) of ...;
type A_2 is array (ST range <>) of ...
subtype NR is ST range -49..-50;  -- see later discussion
X: A_1(NR);       -- no CONSTRAINT_ERROR; -49 in INTEGER;
Y: A_2(NR);       -- CONSTRAINT_ERROR?; -49 not in ST
```

The declaration of Y does not raise CONSTRAINT_ERROR according to the current wording of the LRM, since compatibility is not explicitly defined for null index ranges. (The intent, however, seems clearly to be that CONSTRAINT_ERROR would be raised when Y's declaration is elaborated.)

In short, the current wording does not properly separate the conditions under which CONSTRAINT_ERROR is raised for a discrete_range and the conditions under which CONSTRAINT_ERROR is raised for a discrete range <u>used</u> as an index range.

Note that because of the special restriction on null discrete_ranges (or null index ranges), similar-appearing constructs in the language sometimes do not both raise CONSTRAINT_ERROR:

```
subtype N is INTEGER range 1..-10;
subtype M is LOW range 50..49;
```

Neither of these declarations raises CONSTRAINT_ERROR, since the only requirement for a null range (see 3.5.0) as opposed to a null index range is that the bounds be values of the base type, which in this case is INTEGER. However, if subtype M or N are used as discrete ranges in an index constraint for T or A_2, then CONSTRAINT_ERROR will be raised, since in the first case, the upper bound is not the predecessor of the lower bound and in the second case, the lower bound does not belong to the subtype range defined by T's index subtype, LOW, or A_2's index subtype, ST. Note that M and N can be used as subtype indications in a membership operation or an object declaration without raising CONSTRAINT_ERROR.

3.6.1.b Index Constraints

Compile-time Constraints

1. An index constraint can only be applied to a type_mark of an array type or of an access type designating an array type.

2. One discrete_range must be provided in an index constraint for every index of the array type being constrained.

3. The base type of each discrete_range in the index constraint must be the same as the base type of the array index for which the discrete_range applies.

4. An index constraint must not be given in a subtype_indication if the type_mark denotes a previously constrained array type.

5. An index constraint must not be given in an allocator for an access type previously constrained with an index constraint.

6. An index constraint must be given in a subtype_indication if the type_mark denotes an unconstrained array type and the subtype indication is used in a (non-constant) object_declaration, derived_type_definition, component_declaration, or generic_actual_parameter corresponding to a constrained array formal parameter.

7. An index constraint must be given in an array_type_definition used in a component_declaration.

8. An index constraint must be given in an allocator containing an unconstrained array type_mark if no initial value is specified for the allocated array (see 4.8.S).

Exceptions

1. CONSTRAINT_ERROR is raised if a non-null discrete_range in an index constraint specifies a range of values L through R, the type_mark of the corresponding array index is T, and T'FIRST > L or T'LAST < R. (This check is suppressed by applying RANGE_CHECK to the array type or to T [or to the access type designating the array type being constrained?].)

2. CONSTRAINT_ERROR is raised if the lower bound of a null discrete range in an index constraint equals L, the type_mark of the corresponding array index is T, and T'FIRST > L (see 3.6.1.a.G/2). (This check is suppressed by applying RANGE_CHECK to the array type or to T [or to the access type designating the array type being constrained?].)

Test Objectives and Design Guidelines

1. Check that

   - an index constraint cannot be applied to a scalar type, record type, private type implemented as an array type, or an access type designating any of these types.

   - the number of discrete ranges in an index constraint cannot be less than or greater than the number of indexes in the array type being constrained. Check for both array and access types.

   - the base types of the discrete ranges cannot be different from the base types of the corresponding indexes.

   - an index constraint cannot be given in a subtype indication whose type mark denotes a constrained array type.

   - an index constraint cannot be given in an allocator for an access type designating a constrained array type (see 4.8.T).

   - an index constraint cannot be omitted in a subtype indication whose type_mark denotes an unconstrained array type if the subtype indication is used in a non-constant object_declaration, an array_type_definition (used in a component_declaration generic_type_definition, type_definition (see Gaps), object_declaration; see Gaps), or a generic_actual_parameter corresponding to a constrained array type.

   - an index constraint cannot be omitted in an allocator naming an unconstrained array type if no initial value is specified for the array being allocated.

   - an index constraint must contain at least one discrete range.

   - an index constraint cannot contain a box symbol.

2. Check that CONSTRAINT_ERROR is raised if one of the bounds of a non-null discrete range does not lie in the range of the type_mark for the index being constrained.

   Check that CONSTRAINT_ERROR is raised if the lower bound of a null index constraint does not lie in the range of the type_mark being constrained.

   Check that CONSTRAINT_ERROR is raised if the upper bound of a null range is not the predecessor of the lower bound.

Implementation Guideline: Check all the above conditions for discrete ranges having each of the permitted forms: ST, ST range L .. R, L .. R, and A'RANGE.

3. Check that an index constraint can be omitted when declaring a formal parameter of a subprogram or generic unit, in which case, the bounds of the formal parameter are defined when the subprogram is invoked or the generic unit is instantiated.

   Check that if an index constraint is provided for a formal parameter of a subprogram or generic unit, the actual parameter must have the same index bounds as the formal parameter or else CONSTRAINT_ERROR is raised.

4. Check that an index constraint can be omitted when declaring a constant object with a subtype_indication and that the bounds of the object are determined by the bounds of the initial value.

5. Check that an index constraint can be specified in an allocator naming an unconstrained array type (see 4.8.T).

6. Check that an index constraint in a record component declaration can give the name of a discriminant of the record. Check for record components declared with a subtype_indication, for the index bounds of an array declared with an array_type_definition, and for the bounds of the component of an array specified with an array_type_definition (see 3.7.2.T).

Gaps

1. The LRM contains the following statement:

> [If an array subtype indication] contains the type mark of an unconstrained array type, index bounds must be specified by an explicit index constraint in variable declarations and in the subtype of components; the index constraint can be omitted from the declaration of a constant, in which case, the bounds are those of the initial value.

A careful reading of this statement might suggest that the following declarations are permitted:

```
MONEY: constant array (1..3) of STRING := ("PENNY", "NICKEL",
 "DIME");
type T is array (1..3) of STRING;
MONIES: constant T := ("PENNY", "NICKEL", "DIME")
ST: constant T := ("AB", "CD", "DE");
```

In the first two cases, each array component has a different length and the type T can only be used to declare constant objects. In the third case, all three components have the same length.

The justification for saying these declarations are permitted is the

phrase specifying when an index constraint can be omitted from an object declaration. The phrase does not clearly distinguish bstween an index constraint specifying the bounds of the object, as in:

```
NAME: STRING := "JON";  -- NAME'LAST = 3
```

and the index constraint used to specify the bounds of the array component (rather than the array containing the component). We suspect that the intent was to permit the omission of an index constraint only in the form of constant object_declarations syntactically specified with a subtype_indication.

The LRM could be clarified by saying "the index constraint can be omitted from the declaration of a constant object or component of a constant object" (which would permit the examples in question), or by saying "the index constraint can be omitted from the declaration of a constant object but not from the declaration of a component of an array type" (which would forbid the examples).

3.6.2 Array Attributes

Semantic Ramifications

Note that attributes are defined for objects as well as for types (unlike the scalar attributes 'FIRST and'LAST).

Note that 'FIRST and 'LAST produce results whose type is that of the index, whereas 'LENGTH produces a result of a (predefined) integer type. Hence, A'LENGTH = A'LAST - A'FIRST + 1 is a legal Ada expression only if A's index type is an integer type.

Since 'LENGTH produces a result of an integer type, if the 'LENGTH value is required to be of integer type T and its value is greater than T'LAST, CONSTRAINT_ERROR is raised. However, one must keep in mind the rules for determining the type of a result in various contexts, e.g., consider:

```
type T is 1..6;
A: array (1..10) of INTEGER;
L: T := T'LAST;
B: BOOLEAN := A'LENGTH = L;
```

CONSTRAINT_ERROR is not raised when B's declaration is elaborated, since the derived type rules imply that "=" takes operands of type T'BASE in this case, and A'LENGTH does not exceed the value of T'BASE'LAST. Hence, B is declared with an initial value FALSE.

Note that since A'RANGE defines a (scalar) subtype, A'RANGE'FIRST is a permitted attribute name. Moreover, A'RANGE can be used in object declarations, e.g.,

```
type A is array (1..10) of T;
X: A'RANGE:
```

The declaration of X is equivalent to:

```
X:INTEGER range 1..10;
```

Note that since 'LENGTH is overloaded, an expression of the form A'LENGTH = 6 will be ambiguous (see 6.6), and hence illegal if more than one integer type is available, e.g., if an implementation supports both an INTEGER and a LONG_INTEGER type, or if a derived integer type has been declared:

```
type T is 1..100;
X: BOOLEAN := A'LENGTH = 6;      -- ambiguous
```

Unlike in index ranges, there is no assumption that an integer literal is of type INTEGER in the absence of any additional information.

Note that these attributes can be applied to formal parameters, and in particular, to unconstrained formal array parameters:

```
type T is array (INTEGER range <>) of INTEGER;
A: T(1..50);
procedure P (X: T);
```

Within P, one can write X'FIRST, X'LAST, and X'RANGE, etc. For the call P(A), these would yield the values 1, 50, and INTEGER range 1..50. For a slice, P (A(5..10)), the values would be 5, 10, and INTEGER range 5..10, and similarly for a null slice.

Compile-time Constraints

1. An attribute of the form 'FIRST(N), 'LAST(N), 'LENGTH(N), 'RANGE(N), 'LENGTH, or 'RANGE can only be applied to a constrained array type or an object of an array type.

2. An attribute of the form FIRST or LAST can only be applied to an array type, a scalar type, or an object of an array type.

3. The attributes FIRST, LAST, LENGTH, and RANGE can have at most one argument, and that argument must be a static expression of an integer type having a value greater than zero and less than the number of dimensions defined for the array type.

Exceptions

1. CONSTRAINT_ERROR is raised if the LENGTH attribute yields a value of type T and this value is greater than T'LAST. (This exception is suppressed by RANGE_CHECK for the type T.)

Test Objectives and Design Guidelines

1. Check that

   - the array attributes cannot be applied to an unconstrained array type.
   - the parameterized attribute forms cannot be applied to a scalar type or scalar object;
   - the attributes cannot be applied to a record or private type.
   - more than one argument is not permitted.
   - the attributes' argument cannot be a non-static expression.
   - the value of the atrributes' argument cannot be zero, negative, or greater than the number of dimensions of the relevant array type.
   - 'LENGTH compared to integer literal is considered ambiguous in the presence of more than one integer type.

2. Check that CONSTRAINT_ERROR is raised if A'LENGTH yields a value of type T and this value exceeds T'LAST.

   Implementation Guideline: Declare an array type A, for example, with bounds -1..INTEGER'LAST and check A'LENGTH in a context requiring an INTEGER value, e.g.,

   ```
   X := A'LENGTH - 2;
   ```

   where X is of type INTEGER.

3. Check that CONSTRAINT_ERROR is not improperly raised for a derived type, e.g.,

   ```
   type T is new INTEGER range -1..10;
   type U is array (T) of INTEGER;
   X: T := U'LENGTH - 2;
   B: BOOLEAN := U'LENGTH > X;
   ```

4. Check that each of the parameterized and unparameterized array attributes yield the correct values or subtype when applied to an array type or an array object.

   Check that the RANGE attribute can be used to declare objects.

5. Check that the attributes give the appropriate values when applied to an unconstrained formal parameter. In particular, check that when the actual parameter is a slice (including a null slice), the appropriate results are produced.

Gaps

1. If an implementation only supports an INTEGER type and there are no visible objects of a derived integer type, it is unclear whether A'LENGTH = 6 would be considered ambiguous:

```
package P is
    type T is 1..100;
end P;
A: array (1..10) of INTEGER;
B: P.T;
C: BOOLEAN := A'LENGTH = 6;
```

If the declaration of B is removed, is A'LENGTH = 6 ambiguous?

3.6.3 Strings

Semantic Ramifications

Because string literals are considered a positional aggregate of CHARACTERs, they are implicitly indexed starting with NATURAL'FIRST (see 4.3.2), e.g., given

```
procedure P(X: STRING);
```

if we call P with the argument "ABC", then X'FIRST = 1 and X'LAST = 3. Similarly, if we initialize a constant with a string literal:

```
C: constant STRING := "ABC";
```

C'FIRST = 1 and C'LAST = 3.

We assume that a null string literal has the bounds 1..0 (see Gaps), and hence within P(""), X'FIRST = 1 and X'LAST = 0.

Note that any one-dimensional array has string-like properties, since concatenation and lexicographic ordering relations are predefined for such types. In addition, for one-dimensional arrays of an enumeration type containing at least one character literal, string literals are defined using the characters in the set, as illustrated in the LRM for the type ROMAN_DIGIT (see 3.5.2).

There is no requirement that string types all have a lower bound of one.

```
subtype STRANGE is INTEGER range INTEGER'LAST -50..INTEGER'LAST;
type BIZARRE is array (STRANGE range <>) of ROMAN_DIGIT;
X: constant BIZARRE := "XXI";
```

X'FIRST = STRANGE'FIRST and X'LAST = STRANGE'FIRST + 2. It is not even required that STRING variables have a lower bound of one:

```
        STRANGER: STRING (INTEGER'LAST-3..INTEGER'LAST) := "ABCD";
```

If we now call P(STRANGER), within P, X'LAST = INTEGER'LAST.

In short, just because STRING is a predefined type does not imply any special constraints placed on its use other than those implied by its definition.

Since concatenation, the relational operators, and the membership operators are predefined for all one-dimensional arrays, the declaration of such a type produces (see Gaps) implicit declarations of these operators in the context containing the one-dimensional array type declaration. Hence, if the type BIZARRE was declared in package P, we can reference the P."&" and P."=", etc. Such naming capabilities are also, of course, permitted for anonymous one-dimensional array declarations, e.g.,

```
        X: array (1..10) of CHARACTER;
```

X is not assignable to any STRING variable, but it does imply local definitions of the predefined one-dimensional array operators.

Test Objectives and Design Guidelines

1. Check that the predefined STRING type conforms to the specified definition, i.e., that NATURAL'FIRST = 1, NATURAL'LAST = INTEGER'LAST and that STRING subtypes can be declared with the range 1 .. INTEGER'LAST.

2. Check that a STRING VARIABLE can be declared that is indexed with a value greater than one, e.g., 5..10 or INTEGER'LAST-5..INTEGER'LAST.

3. Check that the lower bound of a null STRING literal is one and the upper bound is zero.

   Check that a non-null STRING literal has a lower bound of one and an upper bound of its length minus one.

4. Check that constant STRING objects can be declared where the bounds are determined by the initial value.

5. Check that a STRING variable is considered an array, i.e., the array attributes can be applied to such a variable.

6. Check that string literals can be formed for character and string types other than the predefined CHARACTER and STRING type, and that the capabilities tested in the previous tests are available for such user-defined types.

7. Check that the predefined operations for STRING are correctly implemented and that these operations are available for any one-dimensional array type (see 4.5.2.T and 4.5.3.T)

8. Check that when a one-dimensional array type is declared in a package, the

predefined operators can be named using the package name as a prefix and that these operators are not available outside the package unless they are directly visible (see Gaps) (see 3.4.T/5,6).

Gaps

1. The LRM does not explicitly define the bounds of a null string literal. Since there are no null positional arrays, the LRM's definition that a string literal is "a special form of positional aggregate" does not imply the lower bound of "" is 1. If "" were defined as (2..1 => ' '), its lower bound would be 2.

   The only time when the question of the null string's bounds has a detectable semantic effect is when a null string is passed to an unconstrained string formal parameter and the value of 'FIRST for the formal parameter is used. Nonetheless, the LRM should standardize what the lower bound of a null string is expected to be, e.g., it should be the lower bound of its base type's index subtype. Hence, the 'FIRST attribute for a null string of type BIZARRE would be INTEGER'LAST-50.

2. Our conclusion that the declaration of a one-dimensional array type implies the local implicit declaration of predefined operators for the type follows from the rules that apply to derived types (see 3.4.S). However, strictly speaking, an array type declaration that uses an array_type_definition is not a derived type declaration (unlike the case for integer type definitions). However, for uniformity, we assume that the predefined operations applicable to a one-dimensional array are implicitly declared just as for derived types and that these operations can be redefined (e.g., we could provide a special definition of "&" for the BIZARRE one-dimensional array type).

   The only situation in which one might want to actually name these implicitly declared operators is in a renaming declaration given outside the context in which the array type declaration was introduced, e.g.,

```
package P is
    type BIZARRE is ... ;
end P;
function "&"(X,Y: BIZARRE) return BIZARRE renames P."&";
X: constant P.BIZARRE := "XX" & "I";
```

   Without such a renaming, it would seem that the above use of the concatentation operator would be illegal, since no "&" operator would be visible that would produce a BIZARRE array result.

   Another interpretation could be that the predefined operators for one-dimensional arrays are always visible in the absence of any superseding declaration, hence implying that the above declaration of X would be legal even without the renaming declaration. However, this would be a different effect from that implied for integer and floating point types, for example.

3.7 Record Types

Semantic Ramifications

3.6.1 states that an array object declared as a record component must have either static bounds or bounds that are specified with the name of a discriminant. This implies that the following record component declarations are illegal:

```
X: INTEGER := 5;
subtype S is INTEGER range 1..X;
type T is
    record
        A: array (1..X) of INTEGER:      -- illegal
        B: array (S) of INTEGER;         -- illegal
```

The declaration of A is illegal since X is not a static expression. The declaration of B is illegal since S is not a static subtype, i.e., its range constraints are not specified as static expressions.

In addition, there is a restriction stated in 3.7.1 that the only permitted dependences between record components are those that involve certain uses of discriminant names. Hence, continuing the declaration of type T, the following component declaration would be illegal:

```
C: array (A'FIRST..5) of INTEGER;       -- illegal
```

Even though A'FIRST is a static value, the declaration of C is not permitted to depend on an attribute of another component. It is not forbidden, however, for an initial value of a component to depend on an attribute of itself, e.g., continuing the declaration of T:

```
D: INTEGER range 1..X := D'LAST;
```

since only dependences between components are forbidden.

Although the initializing expression for a component declaration is evaluated when the record type declaration is elaborated, its validity as a default value cannot be determined when an index bound or discriminant of the component depends on the value of a discriminant:

```
type R (M: INTEGER range 3..4) is
    record
        case M is
            when 3 => S: INTEGER range 1..10;
            when 4 => T: INTEGER range 11..20;
        end case
    end record;
R3_10: constant R := (L => 3, S => 10);
```

```
type T (L: INTEGER := 3) is
    record
        DATA: STRING (1..L) := "ABC";
        DATUM: R(L) := R3_10
    end record

A: T(3);          -- default initial values are ok
B: T(4) := (DATA => "ABC", DATUM => (L => 4, T => 11));
C: T(4);          -- CONSTRAINT_ERROR raised
D: T;             -- default initial values are ok
```

Only the declaration of C raises CONSTRAINT_ERROR, since only in this case is an invalid default value to be assigned to both DATA and DATUM. In DATA's case, the default value is too short, and in DDATUM's case, the default discriminant value is not 4. Hence, although the default value is a proper value of DATUM's base type (and hence, is a legal value when T's declaration is elaborated), it is only a valid default initial value when L equals 3. Note that CONSTRAINT_ERROR is not raised for B's declaration, since 3.2 says default values are not assigned if an explicit initialization is given.

The LRM rule that says CONSTRAINT_ERROR is raised for C's declaration is given in 3.2 -- "Elaboration of an object declaration (i.e., C's declaration) with ... a default initialization raises the *exception* CONSTRAINT_ERROR if the initial value fails to satisfy some constraint on the object." In C's case, the constraint on the object is T's discriminant constraint, whose value is 4. Note that in D's case, the default discriminant constraint value is 3. Since this default value is compatible with the default values specified for T's components, no CONSTRAINT_ERROR is raised.

In short, although the rule in 3.7 implies CONSTRAINT_ERROR due to default initialization is raised when a record type definition is elaborated, the rule in 3.2 recognizes that there are some cases where insufficient information exists to decide whether CONSTRAINT_ERROR should be raised. These are precisely those cases covered by the rule in 3.2 -- when a constraint specified for an object determines the validity of a default initial value. Since records are the only types for which default values can be specified, the rule in 3.2 for default values applies only to record components having default values and whose type depends on a discriminant.

Note also that the DATA component of T could not have been declared as:

```
DATA: STRING (1..L) := (DATA'RANGE => ' ');
```

because DATA'RANGE depends on the value of L, and the use of a discriminant in determining an initial value is not one of the dependences between record components permitted by 3.7.1.

The rule specifying when CONSTRAINT_ERROR is raised for default initializations says that a default value "must satisfy any constraint" imposed on the component. The rule does not say the constraint must be satisfied "as for assignment". The omission of "as for assignment" has the following implications:

```
type T is array (1..5) of INTEGER;
type R is
    record
        A: T := (2..6 => 0);     -- CONSTRAINT_ERROR raised
    end record;
```

The index constraint imposed on A by T is 1..5. The bounds of an array value only satisfy an index constraint "if they are equal to the bounds of the index constraint" (3.6.1). Since the bounds of the default value are 2..6, the index constraint is not satisfied and CONSTRAINT_ERROR is raised. Note, however, that if an object of type T were declared and explicitly initialized:

```
X: T := (2..6 => 0);
```

no exception would be raised, because 3.2 (rule b) says that <u>explicit</u> initializations "must satisfy any constraint on the [initialized] objects as for assignment statements." Satisfying an index constraint "as for assignment" means checking that corresponding dimensions of the assigned value and the index constraint have the same number of index values (see 5.2.1). Hence, the initialization of X raises no exception.

We do not believe this difference between the rules for explicit and default initializtion is an oversight in the LRM, because initializing a record component is essentially the same as providing a value for that component in a record aggregate. The rules for aggregates (4.3) say that CONSTRAINT_ERROR is raised if the value given for a component does not "satisfy any constraint associated with the ... component." This is the same as the rule for default initializations, and default initializations, after all, are giving values to components.

For example, it would seem anomalous to permit (2..6 => 0) as a default initialization for component A, when this aggregate could not be written in an explicit initialization for objects of type R, e.g.

```
Y : R := (A => (2..6 => 0));
```

CONSTRAINT_ERROR is raised when the aggregate is evaluated. (Note: the rule for aggregates exists to make overloading resolution of aggregates easier.)

With respect to suppressing exceptions, checking that a specified range of index values satisfies an index constraint is associated with INDEX_CHECK, not RANGE_CHECK (see 1.7).

<u>Compile-time Constraints</u>

1. No duplicates are permitted among the identifiers declared in a set of component_declarations (see also 3.7.3.C/1 and 3.7.2.C/5).

2. If a component_declaration contains an array_type_definition, the array type must specified with an index_constraint.

3. An array subtype_indication in a component_declaration must specify a constrained array.

4. In a record component declaration, every bound of an index constraint in a subtype indication or array type definition must be specified either with a static expression or with the name of a discriminant of the enclosing record.

5. A component list cannot be vacuous, i.e., it must contain at last one component_declaration, at least a variant_part, or at least the reserved word null (see Gaps).

6. No dependences between record components are permitted except the use of a discriminant as a bound in an index_constraint, or as the discriminant_name of a variant_part, or to specify a discriminant value in a discriminant specification.

7. The base type of a record component and an initial value must be the same.

8. A default initial value must not be specified for a component if assignment is not available for the component's type, e.g., the component is of a task type, limited private type, or composite type for which assignment is not available.

Exceptions

1. CONSTRAINT_ERROR is raised when the initial value specified for a scalar component, C, does not lie in the range C'FIRST..C'LAST. (This check is suppressed by RANGE_CHECK applied to the component type.)

2. CONSTRAINT_ERROR is raised when an initial value is specified for an array component whose index constraint does not depend on a discriminant, if the index bounds of the initial value are not equal to the bounds of the index constraint. (This check is suppressed for a given index if INDEX_CHECK is suppressed for the index's type. It is suppressed for every index if INDEX_CHECK is suppressed for the array type.)

3. CONSTRAINT_ERROR is raised when an initial value is specified for a component whose subtype indication imposes a discriminant constraint:

   - if the imposed discriminant constaint does not depend on a discriminant of the enclosing record; and

   - each discriminant of the initial value has the value imposed by the corresponding discriminant specification (see 3.7.2).

   (This check is suppressed for a given discriminant if INDEX_CHECK is suppressed for the discriminant's type. It is suppressed for every discriminant if INDEX_CHECK is suppressed for the component's type.)

Test Objectives and Design Guidelines

1. Check that within a record type definition, duplicate record component identifiers are not permitted, either within a component declaration, between the component declarations preceding a variant_part, within a

variant_part, between variant_parts, between a variant_part and any preceding component declaration, or between the names of discriminants and any record component name (see 3.7.2) (see 8.3.c.T/1).

2. Index constraints can be used in the subtype_type indication of a component_declaration or with an array_type_definition to constrain the array type or the components of the array type. Check that no index constraint bound can be specified with a non-static expression (see 4.9.S) or with an expression using the name of another component of the same record, including attribute names.

   Implementation Guideline: Do not use the name of a discriminant in this test; see the next test.

3. Check that an expression containing more than one primary such that the name of a discriminant is one of its primaries, cannot be used to specify the bound of an index, the value of a discriminant, the bound of a range in a range constraint or accuracy constraint, a default initial value (or a choice in an aggregate containing a single component association [see 4.3.2]), nor can such an expression contain just the name of a discriminant if the expression is used in an indexed component (e.g., if A is the name of a previously declared constant array and L is the name of a discriminant, A(L) cannot be used in any expression within the record definition.)

   Implementation Guideline: Use expressions in specifying the component type of an array_type_definition, as well as in the array type definition's index constraint.

4. Check that

   - an unconstrained array cannot be declared as a component type using either an array_type_definition or a type_mark for an unconstrained array type (even if it is initialized with a static value).

   - neither the name of a component of the record being defined nor an attribute of such a component nor an attribute of a discriminant (see 3.7.1.S) can be used in specifying an index constraint, discriminant constraint, accuracy constraint, range constraint, or initial value of another component of the record.

   - the name of the component being declared cannot be used to specify any constraint associated with the subtype indication or array type definition being used to declare the type of this component;

   - the base types of a component and its initial value cannot be different;

   - a default initial value cannot be specified for a component of a task type, limited private type, or composite type having a component for which assignment is not available.

- a vacuous component_list is forbidden.

- if different components are declared with identical array type definitions, they are not of the same type and cannot be assigned to each other.

- the name of the record type being declared cannot be used within the record to form an attribute name for the record or a component of the record.

5. Check that for a component of a scalar type, a non-static expression can be used to specify its range constraint or default initial value.

6. For a component of a record or private type, check that a non-static expression can be used in a discriminant constraint or in specifying a default initial value.

   Check that for a component of a limited private type, a non-static expression can be used in a discriminant constraint for a component.

7. For a component of an array type (whether declared with an array type definition or a subtype indication), check that a non-static expression can be used in an aggregate specifying an initial value for the component.

   Check that the initializing value can be specified with a one component aggregate having a non-static choice (see 4.3).

8. Check that CONSTRAINT_ERROR is raised if an invalid value is specified as the default initial value for a scalar, record, array, or access type whose constraint does not depend on a discriminant.

9. Check that an unconstrained record type can be declared as a record component and initialized with an appropriate value.

10. Check that CONSTRAINT_ERROR is raised in an object declaration if a default initial value has been specified for a record or array type whose constraint depends on a discriminant, and no explicit initialization is provided for the object.

    Check that CONSTRAINT_ERROR is not raised in the above case if the default initial value satisfies the constraint associated with the value of the discriminant given either explicitly or by default in the object declaration.

11. Check that there exist default initial values for a component of an array type that raise CONSTRAINT_ERROR even when the same default value could be assigned to an object of the component's subtype without raising an exception.

    Implementation Guideline: Check both for arrays whose constraint does and does not depend on a discriminant.

12. Check that a record can be declared without a variant part, with only a variant part, and with null.

13. Check that if several components are declared in the same component declaration with an array type definition, the components are considered to be of the same type.

Gaps

1. The last sentence before the example is phrased in a way that suggests (but does not actually state) that if a component_list does not contain either a component declaration or a variant part, it must contain the word null. For example, under 3.7.3, it is stated that "A variant can have an empty component list, which must be specified by null." No similar statement appears for component_lists in record type definitions, but it would be inconsistent with the requirement for null elsewere in the language for the requirement to be omitted here. So we have assumed that the absence of an explicit requirement for null in the component_list of a record_type_definition is an oversight.

3.7.1 Discriminants

Semantic Ramifications

Note that since discriminants are considered record components, the rule prohibiting dependences among record components applies to discriminants as well. In particular, attributes of discriminants cannot be used to specify the value of an index constraint, initial value, etc.

The discriminants of a record object cannot be changed, even by a complete record assignment, if the type of the object is constrained (see 3.7.2).

Both limited and non-limited private types can be declared with discriminants.

The role of default discriminants is discussed in 3.7.2.S.

The LRM states that discriminants "[appear] before any of the components in the [record] type definition" so as to define the order in which values must appear in record aggregates of a type having discriminants, namely, the discriminant values must be given first in a positional aggregate. There is no requirement that discriminants in a record representation occupy the initial locations of the record. Note however, that the presence of a discriminant is independent of whether or not it is used in a variant part. No matter what variants are specified, discriminant components can always named (see 3.7.3.S).

Compile-time Constraints

1. The type of each discriminant must be an enumeration or integer type.

2. Default initial values must be provided for all or for none of the discriminants in a discriminant_part.

3. The name of a discriminant can only be used as a bound in an index constraint, as the discriminant_name in a variant_part, or to specify a value in a discriminant.

4. Attributes of discriminants cannot be used within a record type definition to specify the value of a required expression.

5. The names specified for discriminants must all be different, and none of the record component names specified in a record type definition having discriminants can be the same as a discriminant name (see 8.3).

6. A discriminant of a record object cannot be named as a variable in an assignment statement (see 5.2) or used as an actual in out or out parameter.

Exceptions

Exceptions can be raised when the subtype indication is elaborated or the expression defining the default initial value is evaluated.

Test Objectives and Design Guidelines

1. Check that

- discriminants cannot be a fixed or floating point type, nor a composite, private, limited private, or task type.

  Implementation Guideline: For the private and limited private cases, be sure the private type is implemented as a discrete type.

- default initial values cannot be provided for only some discriminants.

- the name of a discriminant can only be used as an index constraint bound, a discriminant name in a variant part, or to specify the value in a discriminant (see 3.7.T/3 and 3.7.3.T/1).

- attributes of discriminants cannot be used within a record type definition (see 3.7.T/4).

- the names of discriminants in a discriminant_part must all be different and cannot be the same as the names of any of the record components in the following record type definition (see 8.3.c.T/1).

- direct assignments to disriminant components are forbidden (see 5.2.T).

- a discriminant component cannot be named as an actual <u>in</u> <u>out</u> or <u>out</u> parameter (see 6.4.1.T/1).

2. Check that values of discriminant components can be accessed in objects of a type declared with discriminants (see 4.1.3/T).

3. Check that the following types are permitted as the type of a discriminant:

   - BOOLEAN

   - CHARACTER

   - user-defined enumeration type

   - INTEGER

   - user-defined types derived from these types

   <u>Implementation Guideline</u>: Use an integer type, an enumeration type, and a derived discrete type in forming the derived types for this test.

4. Check that a private type implemented with a discrete type can be used as a discrete type within the scope of the declaration implementing the type.

5. Check that a record consisting only of discriminant components can be declared.

<u>3.7.2 Discriminant Constraints</u>

<u>Semantic Ramifications</u>

For an unconstrained record private, or limited private type_mark, T, declared with discriminants having no default initial values, the following contexts are the only ones in which T is permitted in a subtype indication without a discriminant constraint:

- a subtype declaration;

- a formal parameter declaration a subprogram or generic unit;

- as the second operand of a membership operator (see 4.5.2.S);

- an access_type_definition;

- as an actual generic parameter corresponding to a generic formal type declaration with a discriminant part.

The contexts in which T requires a discriminant constraint in a subtype indication are:

- an object declaration.
- an array_type_definition (to specify the component type) (see 3.6.);
- a record component declaration;
- an allocator when the object being allocated is to be uninitialized except for its discriminants (see 4.8);
- a generic actual parameter corresponding to a private type declaration without a discriminant part;

If a discriminant part has no default initial values, all objects of that type are created with fixed discriminant values that cannot be changed by assignment. The presence of default discriminants implies that objects can be created whose discriminant values are not fixed; they can be changed by assignment to the entire object. In addition, the default values ensure that every record object has defined discriminant values, whether these are explicitly provided by a discriminant constraint (or for access types, in an initial value for the entire record), provided by default in the absence of a discriminant constraint, or provided by an actual parameter.

Whether objects having default discriminants are created with frozen or with changeable discriminant values is reflected by the value of the 'CONSTRAINED attribute for the object. This attribute is needed particularly when assigning to unconstrained formal paameters, e.g.,

```
type T (L: INTEGER) is record ... end record;
CONS_1, CONS_2 : T(5);
UNCONS_1, UNCONS_2 : T;
procedure P (X: T) is
begin
    X := UNCONS_1;
end P;
```

If we call P(CONS_1), the assignment to X raises CONSTRAINT_ERROR if the value of UNCONS_1.L is not 5; since CONS_1 is constrained, CONS_1 cannot be assigned a value that would change its discriminant. If we call P (UNCONS_2), then CONSTRAINT_ERROR will not be raised by the assignment to X, since the corresponding actual parameter does not have a fixed discriminant value.

In the call P(CONS_1), X'CONSTRAINED is TRUE, implying no assignement can be made to the actual parameter that will change its discriminant value. In the call (P(UNCONS_1), X'CONSTRAINED is FALSE, and hence no check need be made to see whether the discriminant of the actual parameter is being changed. To summarize, 'CONSTRAINT has the value TRUE for:

- an object or component of an object declared with a discriminant

constraint (whether or not the type of the object has default discriminant values);

- an object designated by an access value, since no assignment to such an object is permitted to change its discriminants (see 3.8), whether or not the type of the object was declared with default discriminant values;

- a formal parameter (of any mode) whose actual parameter's 'CONSTRAINED attribute is TRUE.

'CONSTRAINED has value FALSE for:

- objects declared without a discriminant constraint in an object declaration or in a component of a record or array; such declarations are only legal for record, private, or limited private types with default discriminant values.

There are three main implementation approaches for obtaining the value of 'CONSTRAINED for a given object. If the object is not a formal parameter, the attribute's value can be determined at compile time. If the object is a formal parameter, the actual parameter's 'CONSTRAINED value could be passed as "dope" to the formal parameter; this is necessary only for unconstrained formal parameters of types with default discriminant values. The value of 'CONSTRAINED could also be stored within the object itself, but since 'CONSTRAINED is a property of the object, not its value, an implementation must be careful not to copy the value of 'CONSTRAINED when making assignments between two objects whose 'CONSTRAINED attributes are potentially different, or when comparing the objects for equality. For example, using the variables declared in an earlier example,

```
CONS_2 := CONS_1;
UNCONS_2 := UNCONS_1;
```

these assignments can copy the value of 'CONSTRAINED, since the value of 'CONSTRAINED is the same for the variable being assigned to and the value being assigned. However, copying must <u>not</u> be done for:

```
CONS_2 := UNCONS_1;
UNCONS_2 := CONS_1;
```

since the value of 'CONSTRAINED must not be changed by assignment.

<u>Compile-time Constraints</u>

1. The discriminant names in a discriminant_constraint must only be those of the dicriminant of the type for which the constraint is being specified.

2. A discriminant specification must specify exactly one value for each discriminant (see T/1 for the different cases that must be checked).

3. The base type of a discriminant specified in the discriminant constraint and the corresponding discriminant in a record type must be the same.

4. If a discriminant constraint contains both positional and named specifications, the positional expressions must be given first.

5. A discriminant constraint in a subtype indication can only be given for an unconstrained record or private type, or for an access type designating an unconstrained record or private type.

6. For a record or private type declared without default discriminant values, a discriminant constraint must be specified in every subtype indication naming the type except for subtype indications used to declare formal parameters of subprogram or generic units.

7. The 'CONSTRAINED attribute must only be given for the name of an object (including a formal parameter or component of a composite object) that has discriminants.

Exceptions

1. CONSTRAINT_ERROR is raised (see 3.3) when a discriminant constraint is specified if the value specified for any discriminant does not lie in the permitted range of values for the discriminant. (This check is suppressed by applying RANGE_CHECK to the type being constrained or, for a given discriminant, to the type of the discriminant.)

Test Objectives and Design Guidelines

1. Check that the form of a discriminant constraint is correct, namely, check that:

   - the discriminant_names given in the constraint cannot be different from the names of the discriminants of the type being constrained.

     Implementation Guideline: Check that if a discriminant has been renamed, the new name cannot be used as a discriminant_name in a discriminant constraint (see Gaps).

   - the same name cannot appear twice as a discriminant_name in a particular discriminant_specification or in different discriminant specifications of the same discriminant constraint.

   - if a mixture of positional and named association is used, a named discriminant specification cannot give a value for a discriminant whose value has already been specified positionally.

     Implementation Guideline: Be sure the total number of discriminant values does not exceed the number of discriminants being constrained.

   - too many or too few discriminant values cannot be given.

     Implementation Guideline: Include a discriminant constraint of the form "()"; check for types with and without different discriminant values.

- unnamed (i.e., positional) discriminant values cannot be given after a discriminant_specification using discriminant_names.

  Implementation Guideline: The total number of discriminant values should be correct, and the unnamed values should be in the correct position and of the correct type.

- the base type of the specified discriminant value cannot be different from the base type of the corresponding discriminant.

  Implementation Guideline: Use different discrete types.

2. Check that discriminant_constraints are not permitted where they are forbidden, i.e., check that a discrimininant constraint:

   - cannot be given in a subtype indication for a type mark that has already been constrained;

     Implementation Guideline: Use record, private, limited private, and access types. The imposed discriminant constraint should be identical to the constraint already imposed on the type mark.

   - cannot be specified for a record or private type declared without any discriminants, or for an array, scalar, or task type.

3. Check that a discriminant constraint is not omitted where it is required, i.e., check that discriminant constraint cannot be omitted from a subtype indication for a type_mark having discriminants with no default values when the subtype indication is used in:

   - an object_declaration;

   - an array_type_definition;

   - a record component_declaration;

   - a generic_actual_parameter corresponding to a private type declared without a discriminant part (see 12.3);

   or when T is used in an allocator and an initial value is not specified for the allocated object (see 4.8.T).

4. Check that the 'CONSTRAINED attribute cannot be applied to a record, private, or limited private object whose type has no discriminants, or to an array object.

   Implementation Guideline: Check both for objects designated by an access value and objects declared directly or as formal parameters of subprograms and generic units.

5. Check that if a discriminant of type T is named A, then in a discriminant_specification for the type, A, T.A., etc. can be used as a name for the discriminant.

6. For a type without default discriminant values (but with discriminants), check that an unconstrained type_mark can be used in:

   - a subtype declaration, and the subtype name acts simply as a new name for the unconstrained type;

   - an access_type_definition, and hence a discriminant constraint must be supplied in an allocator that does not provide an initial value (see 4.8.T);

   - an membership operator (see 4.5.2.T);

   - a formal parameter declaration for a subprogram and hence, the constraints of the actual parameter are available within the subprogram, 'CONSTRAINED is TRUE, and assignments to the formal parameter cannot attempt to change the discriminants of the actual parameter without raising CONSTRAINT_ERROR;

     Implementation Guideline: For the actual parameter, use an object designated by an access value as well as other objects.

   - as an actual generic parameter corresponding to a generic (private) type declaration having a discriminant part (see 12.3.2.T).

7. For a type with or without default discriminant values, check that a discriminant constraint can be supplied in the following contexts and has the proper effect:

   - in an object_delcaration, component_declaration or subtype indication of an array_type_definition, and hence, assignments cannot attempt to change the specified discriminant values wthout raising CONSTRAINT_ERROR.

   - in an access_type_definition, and hence, access values of this access type cannot be assigned non-null values designating objects with different discriminant values (see 5.2.T/).

   - in an allocator, and the allocated object has the specified discriminant values (se 4.8.T);

   - in a formal parameter declaration of a subprogram, and hence, assignments to the formal parameter cannot attempt to change the discriminant values without raising CONSTRAINT_ERROR, 'CONSTRAINED is TRUE, and if actual parameters have discriminant values different from the specified ones, CONSTRAINT_ERROR is raised;

   - in a generic actual parameter corresponding to a private type declaration without a discriminant part.

8. For a type with default discriminant values, check that a discriminant constraint can be omitted in:

- an object_declaration, and hence assignments to the object can change its discriminants;

- a component_declaration in a record type definition, and hence assignments to the component can change the value of its discriminants;

- a subtype_indication in an array type definition, and hence assignments to one of the components can change their discriminant values;

- a formal parameter and hence, for parameters of all modes, the 'CONSTRAINED attribute of the actual parameter, becomes the 'CONSTRAINED attribute of the formal parameter, and, for <u>in</u> <u>out</u> and <u>out</u> parameters, if the 'CONSTRAINED attribute is FALSE, assignments to the formal parameters can change the discriminants of the actual parameter; if the 'CONSTRAINED attribute is TRUE, assignments that attempt to change the discriminants of the actual parameter raise CONSTRAINT_ERROR.

9. Check that when assigning to a constrained or unconstrained object of a type declared with default discriminants, the assignment does not change the 'CONSTRAINED value associated with the object assigned to.

   <u>Implementation Guideline</u>: The assignments should use values whose CONSTRAINED attribute's value differs from that of the object being assigned to.

<u>Gaps</u>

1. It is unclear whether the new name for a renamed discriminant can be used as a discriminant_name in a discriminant constraint. We have assumed that such a use of a new name was not intended (see 8.5.G/).

<u>3.7.3 Variant Part</u>

<u>Semantic Ramifications</u>

The subtype of the discriminant_name in a variant_part is given in the corresponding discriminant_part. This subtype is static (see Gaps) if for a subtype_indication of the form ST <u>range</u> L .. R, both L and R are static expressions. If the subtype_indication was merely a type_mark, T, then T could have been declared in a declaration having one of the following forms:

```
type T is range L .. R;        -- 1
type T is (...);               -- 2
type T is new ST;              -- 3
type T is ST range L .. R;     -- 4
subtype T is ST;               -- 5
subtype T is ST range L .. R;  -- 6
```

In form 1, L and R must be static (see 3.5.4), and so T is a static subtype. In form 2, T is a static subtype since it is an enumeration type. In form 3 and 5, T is static if ST is static. In forms 4 and 6, T is static if L and R are both static. Hence, for a type declaration like:

```
type T(L: INTEGER range 1..10) is
    record
        case L is
```

the set of choices need only cover the range 1 through 10. Note that values outside this range can be specified; although the component_lists associated with such values can never be present in objects of the record type, the names of any such component cannot duplicate names given in any other component list belonging to the same record type definition.

If the subtype of the discriminant_name is not static, then the base type of the discriminant specifies the range of values that must be covered (see Gaps).

Note that a variant_part can appear within a variant part, e.g.,

```
case L is
    when 1..5 =>
        case M is
            when 1..10 =>
                etc.
```

Even if the value of discriminant M is used only in a variant part when L is in the range 1..5, space must be allocated for M within the record object no matter what the value of L is for a particular object. Discriminants are components of every object of a type with a discriminant (see 3.7.1).

Note there is no restriction on using the same discriminant name in more than one variant part:

```
case L is
    when 1..5 =>
        case L is
            when 1..2 =>
```

The set of choices to be covered in the inner variant_part is the same as the set in the outer part. Covering just the choice values 1..5 in the inner part would be illegal if the subtype of L was INTEGER range 1..10.

__Compile-time Constraints__

1. The __others__ choice must be present if the set of values included in the set of choices does not cover 1) the set of values associated with the subtype of the discrminant name when the subtype is static; otherwise, 2) the set of values associated with the base type of the discriminant (see Gaps).

2. The base type of the discriminant and each choice must be the same.

3. Every choice must contain only static expressions, i.e., for choices of the form V, L .. R, and ST <u>range</u> L .. R, L, R, and V must be static expressions, and for choices of the form T, where ST is a subtype name, ST'FIRST and ST'LAST must be static expressions (see 4.9).

4. Two choices must not have a value in common.

5. An <u>others</u> choice, if present, must be the only choice given in the last alternative specified for a variant_part.

6. A component list must contain at least one component_declaration, or one variant_part, or <u>null</u>.

<u>Test Objectives and Design Guidelines</u>

1. Check that:

   - the reserved word <u>is</u> is required;

     - <u>when</u> cannot be replaced by <u>if</u>,
       | cannot be replaced by <u>or</u>,
       => cannot be replaced by <u>then</u>,
       <u>end case</u> cannot be replaced by <u>end</u>, <u>endcase</u>, or <u>esac</u>,
       <u>is</u> cannot be replaced by <u>of</u>;

   - the <u>others</u> choice must be the only choice given in the last alternative.

     <u>Implementation Guideline</u>: Try an <u>others</u> choice as the first and middle alternative, and try it as the first, middle, and last choice in a set of choices for the last alternative.

   - a component list cannot be vacuous.

2. Check that

   - the type of the discriminant and each choice must be the same;

   - every pair of choices must cover a disjoint set of values.

   <u>Implementation Guideline</u>: Use both single values and ranges of values, and check for overlapping values with a single alternative and between alternatives. Use overlapping ranges whose end points are different, e.g., 3..5 and 4..6 as well as ranges in which overlap occurs only at the end points. Use some examples in which a large range of values has to be checked for potential overlap. These choices should not all occur in monotonically increasing or decreasing order.

3. Check that non-static choice values are forbidden.

   <u>Implementation Guideline</u>: Try a variable whose range is restricted to a single value. Try a discrete_range of the form T, where ST is a subtype

name having at last one non-static bound, as well as choices of the form ST range L .. R and L .. R, where either L or R is non-static.

4. Check that all forms of choice are permitted in variant_parts statements, and in particular, that forms like ST range L .. R, ST, and A'RANGE(n), where A is an array with static bounds, are permitted.

   Implementation Guideline: Use the name subtype name in more than one choice. For the form ST range L .. R, try at last one subtype name with non-static bounds (see Gaps).

   Check that choices using constant names (including subscripted constant names) are permitted.

5. Check that choices denoting a null range of values are permitted, and that for choices of the form T range L .. R where L > R, neither L nor R need be in the range of ST values.

   Check also that an others alternative can be provided even if all values of the case expression have been covered by preceding alternatives.

   Implementation Guideline: The vacuous alternatives should have null as its component_list in one test and a non-null list in a separate test.

6. Check that choices within and between alternatives can appear in non-monotonic order.

7. Check that relational, membership, and logical operators are allowed as choices only if the expressions containing these operators are enclosed in parentheses.

8. Check that CONSTRAINT_ERROR is raised for non-empty choices of the form ST range L .. R if ST has static or non-static bounds and either L or R is outside ST's bounds (see 3.6.1.T/4).

9. Check that if a discriminant has a static subtype, an others choice can be omitted if all values in the subtype's range are covered, and must not be omitted if one or more of these values are missing.

   Implementation Guideline: Use subtype names with both static and non-static bounds, when possible.

10. Check that if the subtype of a discriminant is not static, others can be omitted if all values in the base type's range are covered, and must not be omitted if one or more of these values are missing.

11. Check that even when the context indicates that a discriminant covers a smaller range of values than permitted by its subtype, an others alternative is required if the subtype value range is not fully covered.

    Implementation Guideline: Use the nested variant_part example at least.

Gaps

1. The definition of a "static subtype" is not given in the LRM.

2. When the LRM says "each value of the discriminant type," we assume it means the discriminant's base type, since the base type of a discrete range always has a statically defined range of values.

3.8 Access Types

Semantic Ramifications

Incomplete type declarations are discussed in 3.8.a.

Note that the term "access object" refers to an object of an access type, i.e., a variable, component, or constant of an access type. Values of an access type are, in essence, pointers. Unlike in some languages, values of an access object (i.e., the pointers) can only point to objects of a specified type.

From an implementation viewpoint, if an access type is declared with a constrained subtype indication, e.g.,

```
type INT_NAME is access INTEGER range 1..100;
```

all variables, constants and components of type INT_NAME either have the value null or point to an integer object whose value is either undefined or in the range 1 through 100. The object pointed to can never be a component of another object, i.e., if A and B are objects of different access types, it can never be the case in non-erroneous programs that an assignment to A.all (the object pointed to by A) changes the value of B.all. Moreover, the objects designated by access values are disjoint from objects that can be named directly, e.g., local or global variables. Hence, if A.all is of type INTEGER and X is a variable of type INTEGER, it can never be the case that an assignment to A.all changes the value of X, and vice versa.

Of course, by use of UNCHECKED_CONVERSION (see 13.10.2) and the 'ADDRESS attribute (see Appendix A), these rules about the independence of access objects of different types and the independence of local objects can be violated. However, an implementation is not required to take such uses of UNCHECKED_CONVERSION into account. It is up to the user of UNCHECKED_CONVERSION to conform to the normal rules of Ada, not vice versa (see 13.10.2). Hence, optimizations can depend on the above statements about how assignments leave the value of certain other objects unchanged.

Different access types may have different representations in two senses (see 13.2). First, the size of the collection associated with the type may be specified explicitly, giving an upper limit on the number of access objects that can be allocated. Second, the number of bits occupied by an access value may be made sufficiently small that in effect, an offset pointer representation must be used. In such a case, the base address for the offset

pointer could be the address of the beginning of the collection, or it might be some address in the middle of the collection, depending on the indexing capabilities of the target machine.

Unless a programmer instantiates the UNCHECKED_DEALLOCATION procedure for an access type (see 13.10.1), there is no operation available for explicitly freeing allocated access objects. If unneeded storage is to be freed, a garbage collector must do it. However, there is no requirement in Ada to provide a garbage collector, and under some circumstances an implementation may choose to provide garbage collection only for certain access types, e.g., access types that are not shared among tasks.

An implementation may choose to statically allocate, i.e., allocate at compile time, access constants or variables that are declared and initialized in the same declaration list where the access type itself is declared, e.g.,

```
procedure P is
   type T;
   type PTR is access T;
   type T is record
         LINK: PTR;
         DATA: INTEGER;
   end record;
   X: PTR := new T(DATA => 0, LINK =>
                        new T(DATA => 1, LINK => null));
begin ... end P;
```

The object designated by X can be allocated on P's stack, since no references to X's object can exist after P is exited -- the type PTR is not known outside of P. If P is a package, X's initial value can be statically allocated at compile time, although care must be taken if PTR is given an offset pointer representation. An implementation might choose to statically allocate X's initial value only if an implementation dependent pragma specifies static allocation for X's initial value.

Note that the syntax and semantics permit, in effect, pointers to pointers, i.e., access types whose objects are in turn access types.

It is important to note that the null access value satisfies all subtype constraints for an access variable, e.g., given

```
JOHN: PERSON_NAME (M) := null;
MARY: PERSON_NAME (F):= null;
```

the assignment JOHN := MARY is legal as long as MARY has the value null. This point is discussesd further in 5.2.

In aggregates, for each component_association, the associated expression (even if it applies to several choices, or to choices of the form others, T'RANGE, etc.) is evaluated only once (see 4.3). Hence, in

```
A: array (1..N) of PTR := (A'RANGE => F());
```

where F is a function returning PTR values, F is evaluated only once, and so, the initial value of each component of A contains the same access value.

For access type declarations in which index or discriminant constraints are imposed as part of the declaration, e.g.,

```
type MALE_NAME is access PERSON(M);
type BOARD_NAME is access MATRIX(8,8);
```

no constraints can be applied subsequently when the type name is used, e.g.,

```
JOHN : MALE_NAME(M);
```

would be illegal. (See 3.7.2 and 3.6.1, which state that a constrained array or record object cannot be further constrained, even if the second constraint imposes the same constraint values as the first).

Note that although declarations of the form:

```
type T (D: DAY) is ... ;
type PTR is access T;
type PTR_PTR is access PTR;
type I is new INTEGER range 1..100;
type INTEGER_NAME is access I;
```

are permitted, the forms:

```
X: INTEGER_NAME range 10..20;
type INT_NAME is access I range 10..20;
```

are not permitted (since only index or discriminant constraints are permitted), nor is the form

```
Y: PTR_PTR(MON);
```

since PTR_PTR does not designate an object having an index or discriminant constraint--it designates an access value of type PTR. Access values do not have discriminants or indices; only records or arrays do. Hence, neither a discriminant constraint nor an index constraint can be applied to an access type whose designated object's type is itself an access type.

Note that normally an unconstrained array type, e.g.,

```
type WORK is array (DAY range <>) of INTEGER;
```

must be used with an index constraint in an object or component declaration, e.g., the following are all illegal:

```
X: WORK;
Y: array (1..10) of WORK;
type T is
   record
```

```
            Z: WORK;
        end record;
```

Similarly, a record type with no default discriminant values, e.g.,

```
        type U(SEX: GENDER) is ...;
```

requires a discriminant constraint (see 3.7.2 and 3.7.3) when used in an object or component declaration. However, no index or discriminant constraint is required when these types are used in access type definitions, e.g., all the following are legal:

```
        type P is access WORK;
        type R is access U;
        P_OBJ: P;
        R_OBJ: R;
```

although constraints can be specified if desired. In all these cases, appropriate discriminant or index constraint values will (i.e., must) be supplied when allocators for P and R are executed, even if U has default constraint values (see 4.8).

When an access subtype indication is used in an access type definition, some care is required in interpreting uses of the resulting type. Consider the following examples, in which PERSON_NAME and PERSON_NAME(M) are access subtype indications:

```
        type PERSON (SEX: GENDER) is record ... end record;
        type PERSON_NAME is access PERSON;
        subtype MALE_NAME is PERSON_NAME(M);
        type PERSON_NAME_PTR is access PERSON_NAME;
        type MALE_NAME_PTR is access PERSON_NAME(M);
        PNP: PERSON_NAME_PTR;
        MNP: MALE_NAME_PTR;
```

Now suppose we execute

```
        PNP := new PERSON_NAME (new PERSON(M));
```

This allocates a pointer, PNP.all, that points to an object whose SEX component is M, i.e., PNP.all.SEX = M. However, the assignments:

```
        PNP := new PERSON_NAME;
        MNP := new PERSON_NAME;
```

allocate pointers whose value is null, i.e., PNP.all = null = MNP.all and hence the component PNP.all.SEX does not exist. Since constraints are not associated with allocated pointer values, but with allocated non-pointer objects, the following sequence of assignments is legal:

```
        PNP := new PERSON_NAME;
        PNP.all := new PERSON(F);
        PNP.all := new PERSON(M);
```

The F and M constraint values are associated with PNP.all.SEX.

But now consider:

```
MNP := new PERSON_NAME;                          -- ok
MNP := new PERSON_NAME(new PERSON(F));           -- CONSTRAINT_ERROR
MNP.all := new PERSON(F);                        -- CONSTRAINT_ERROR
```

The last two assignments raise CONSTRAINT_ERROR, since the constraint on MNP.all implies that MNP.all.SEX must equal M, and these two assignments violate this constraint. However, the first assignment is permitted since after it has been performed, PNP.all = null, and such a value satisfies any constraint.

Compile-time Constraints

1. If a constraint is given in an access subtype_indication (i.e., a subtype_indication whose type_mark is an access type or subtype), the constraint must either be a discriminant constraint or an index_constraint, and the type of the access object must be an unconstrained record or array type with the corresponding discriminants or indices, respectively.

Test Objectives and Design Guidelines

1. Check that record, array, and access type definitions are forbidden in access_type_definitions (e.g., forms like access record ... end record are forbidden).

2. Check that an unconstrained array type or a record type without default discriminants can be used in an access_type_definition without an index or discriminant constraint. Check that (non-static) index or discriminant constraints can subsequently be imposed when the type is used in an object declaration, array component declaration, record component declaration, access type declaration, parameter declaration, derived type definition, allocator (see 4.8.T/6), and return type in a function declaration.

3. Check that if an index or discriminant constraint is provided in an access type definition, the access type name cannot subsequently be used with an index or discriminant constraint in an object declaration, array component declaration record component declaration, access type declaration, parameter declaration, derived type definition, allocator (see 4.8.T/2), or the return type in a function declaration, even if the same constraint values are used.

4. Check that an index or discriminant constraint for an access type can reference a discriminant value in a record component declaration, e.g.,

```
type HSTRING is access STRING;
```

```
type TEXT(L:INTEGER) is
   record
      VALUE: HSTRING(1..L);
   end record;
```

and that slicing and indexing can be applied to such a component.

5. Check that all access objects, including array and record components, are initialized by default with the value null.

6. Check that the object accessed by a constant access object can be modified, e.g., if X: constant PERSON_NAME := MARY, then X.AGE := 39 is permitted and MARY.AGE then equals 39.

7. Check that when initializing an array of access objects with an aggregate containing a single allocator, e.g., (others => new ...), all components of the array point to the same object.

8. Check that an access type used in a subtype indication cannot be constrained with a range constraint or an accuracy constraint.

   Check that a constrained access subtype (i.e., a subtype with an index or discriminant constraint specified) cannot be futher constrained in a subtype indication, even if the same constraint values are used. (Note: this objective differs from T/3 because it refers to access subtype names introduced by a subtype declaration, whereas T/3 refers to constraints specified in an access_type_definition.)

   Implementation Guideline: Try the above checks for access types whose object is an access type as well as for access types whose object is a non-access type.

3.8.a Incomplete Type Declarations

Semantic Ramifications

The type mark declared in an incomplete type declaration must be redeclared later in a regular type declaration. If the incomplete declaration appears in the visible part of a package specification, the full declaration must appear later in the same visible part (i.e., not in the private part) and not in a visible part of an enclosed package, e.g.,

```
package P is
   type T;
   type U;
   package Q is
      type T is access INTEGER;      -- illegal
   end Q;
private;
   type U is access INTEGER; -- illegal
```

Similarly, if the incomplete declaration appears in a private part, the full declaration must appear later in the same private part.

It is apparently intended (see Gaps) that no constraints can be applied to an incompletely declared type mark except a discriminant constraint (if one was given in the incomplete type distribution), e.g., the following is apparently illegal:

```
type T;
type U is access T(1..10);   -- illegal
type T is array (INTEGER range <>) of INTEGER;
```

The statement that "the correspondence between the incomplete and full type declaration follows the same rules as for private types" refers to the rules for correspondence between discriminant parts and the prohibition against a full declaration declaring an unconstrained array type.

There is no rule forbidding the full declaration from declaring a type for which assignment or equality are not available, e.g., a task type, a limited private type, or a composite type having a component for which assignment or equality is unavailable.

Compile-time Constraints

1. If an incomplete type declaration appears in the visible part of a package, the corresponding complete declaration must appear in the same visible part, excluding any nested package specifications.

2. If an incomplete type declaration appears in a private part of a package, the corresponding complete declaration must appear in the same private part, excluding any nested package specifications.

3. If an incomplete type declaration appears in the declarative part of a block, subprogram body, package body, or task body, the corresponding complete declaration must appear in the same declarative part, excluding any nested declarative parts or package specifications.

4. If an incomplete type declaration contains a discriminant part, the corresponding complete declaration must have the same discriminant names, the same subtype indications, and the same default values, all in the same order; the only permitted variation in the lexical structure of the two declarations is that different selected component forms of name are permitted if they denote the same entities in both declarations (see 7.4.1). If an incomplete type declaration does not contain a discriminant part, the full type declaration must not contain a discriminant part or declare an unconstrained array type.

5. [See Gaps] No constraint other than a discriminant constraint can be imposed on a type mark introduced with an incomplete type declaration until the full declaration of the type mark has been elaborated. (After elaborating the full declaration, any constraint compatible with the full declaration can be used.)

6. If a discriminant constraint is imposed on a type mark introduced with an incomplete type declaration, the constraint must be compatible with the discriminants specified in the incomplete declaration (see 3.7.2).

7. Prior to the elaboration of its full declaration, a type mark introduced by an incomplete type declaration can only be used as the type mark in the subtype_indication of an access_type_definition.

Test Objectives and Design Guidelines

1. Check that if an incomplete type declaration appears in the visible or private part of a package, the full declaration must appear in the same part and in particular, 1) cannot be omitted; 2) cannot be given in the private part if the incomplete declaration was in the visible part; 3) cannot appear in the package body's declarative part; 4) cannot appear in a package specification nested in the visible or private part containing the incomplete declaration.

   Check that if an incomplete type declaration appears in the declarative part of a block, subprogram body, package body, or task body, the corresponding complete declaration must appear in the same declarative part, excluding any nested declarative parts or package specifications.

2. Check that an incomplete type declaration can be given for any type, i.e., that the corresponding full declaration can declare an integer type, a real type, an enumeration type, a constrained or unconstrained array type, a record type without discriminants (types with discriminants are checked below), an access type, a task, a private, or a derived type.

3. Check that if an incomplete type is declared with discriminants, the complete declaration must have the same discriminant part, aside from the possible use of selected component notation to disambiguate names (see 7.4.1).

4. Check that prior to its complete declaration, an incompletely declared type:

   - can be used as the type mark in an access type definition;

   - if declared with discriminants, can be used in an access type definition with a compatible discriminant constraint; if not used with a discriminant constraint, check that the declared access type can later be used with compatible discriminant constraints also.

5. Check that prior to its complete declaration, an incompletely declared type:

   - cannot be used in an object declaration, component declaration, or parameter declaration, or as the component type in an array declaration;

   - cannot be used with a range constraint, accuracy constraint, or index constraint in an access type definition, even if the full declaration would permit such constraints;

- an access type declared with the incomplete type cannot be used with index constraints until after the full declaration has been elaborated.

<u>Gaps</u>

1. The specification does not clearly state that the only constraint that can be applied to an incompletely declared type is a discriminant constraint, and that such a constraint must be compatible with the discriminants specified in the incomplete type declaration.

CHAPTER 4

Names and Expressions

4.2 Literals

4.2.a Enumeration Literals

4.2.b Integer Literals

4.2.c Real Literals

Semantic Ramifications

The rounding of real literals according to context is a consequence of the model numbers for types and subtypes (see 3.5.6). Hence appropriate checks are made in the sections of Chapter 3.

4.4 Expressions

Semantic Ramifications

Note that the syntax for expressions containing logical operators does not permit them to be intermixed without using parentheses. Thus, A and B or C is illegal and must be written as (A and B) or C or as A and (B or C). Similarly, unless parentheses are used, sequences of exponentiations, such as A ** B ** C, and sequences of unary operators, such as -+A or A + -B, are not permitted.

Names that denote types, subtypes, subprograms, packages, tasks, entries, exceptions, operators, labels, blocks, or loops are not permitted as primaries in expressions. Similarly, the BASE and RANGE attributes do not have values and hence are not permitted as primaries.

Overloading resolution in expressions is covered in 6.6 and 6.7.

It is important to note that any real expression can be calculated with more precision than that requested. For instance, all computations could be performed double length, with only single length loading and storing to memory. An optimizing compiler is permitted to do some computation at compile-time (with potentially different underlying hardware). These computations are only required to conform to the precision implied by the target machine's real type attributes. Hence, values computed at compile time can differ from those produced at run time for the predefined floating point types and for fixed point types. These points are discussed further in 4.9.c.S.

Compile-time Constraints

1. Only the following names are permitted as primaries in expressions:

   - attributes other than 'BASE and 'RANGE, and

   - names that denote objects (in particular, identifiers declared in object_declarations or number_declarations, indexed_components, slices, selected_components, and function_calls).

Test Objectives and Design Guidelines

1. Check that

   a. the logical operators (and, or, xor, and then, or else) cannot be intermixed in expressions unless parentheses are used to separate the different operators.

      Implementation Guideline: Try at least the following illegal expressions:

      ```
      A and B or C
      A and B and C or else D and E
      ```

   b. a relation can have at most one relational operator, i.e., a sequence of relations such as A <= B <= C is not permitted unless parentheses are used (as in (A <= B) <= C) or the implicit and is made explicit (as in A <= B and B <= C).

   c. unless parentheses are used, a simple_expression can have at most one unary_operator, which must precede the leftmost term.

      Implementation Guideline: Try at least the following illegal expressions:

      ```
      -+A
      - -A
      A + -C
      not not B
      A * -B
      ```

   d. a factor can have at most one exponentiation operator, i.e., a sequence of exponentiations such as A ** B ** C is not permitted unless parentheses are used (as in A ** (B ** C)).

   e. a procedure_call cannot be a primary.

2. Check that

   a. only names that denote objects are permitted as primaries in expressions.

Implementation Guideline: Check that type, subtype, subprogram, package, task, entry, exception, label, block, and loop names are not permitted as primaries.

b. only attributes that have values are permitted as primaries in expressions.

Implementation Guideline: Check that the following attributes are not permitted as primaries: BASE, RANGE.

4.5 Operators and Expression Evaluation

4.5.1 Logical Operators and Short Circuit Control Forms

Test Objectives and Design Guidelines

The precedence of operators and short circuit control forms will be checked in 4.5.0.

Interactions with not are checked here, instead of in 4.5.4.

4.5.1.a Boolean Types

Semantic Ramifications

BOOLEAN types include those derived from the predefined BOOLEAN type. The logical operators can be used with such derived types. However, a derived BOOLEAN type cannot be used as a condition in if statements, etc.

Compile-time Constraints

1. and, and or, and xor are predefined only for Boolean types.

Test Objectives and Design Guidelines

1. Check the correct operation of and, or, and xor, including combinations with not.

   Implementation Guideline: Use deMorgan's law, i.e.,

   not (A and B) = not A or (not B)
   not (A or B) = not A and (not B)
   not (not A and (not B)) = A or B

   as well as simpler tests.

2. Check that non-Boolean arguments to the predefined operators, and, or, and xor, are forbidden.

Check that types derived from BOOLEAN are permitted.

Implementation Guideline: Use integer types and non-discrete types.

3. Check that and then and or else are actually short circuit evaluated.

Implementation Guideline: Use conditions such as

A /= 0 and then B/A > C

or

P /= null and then P.F = D

and check that the appropriate alternative is selected and that no exceptions (such as NUMERIC_ERROR or CONSTRAINT_ERROR) are raised by evaluation of the condition. Try some conditions with and and or operators as well as short circuited and then or or else. First, check out the short circuiting thoroughly with non-nested if_statements, and then try it in all the conditions of a triple nest of if_statements.

4. Check that parenthesized conditions are permitted.

5. Check that

not A and then B

is equivalent to

(not A) and then B

etc.

4.5.1.b Array Types

4.5.2 Relational and Membership Operators

Semantic Ramifications

Operands of relational operators must have the same type, but they may have different subtypes. The subtype of one operand does not have to satisfy the subtype constraint of the other operand. Hence, if X is a variable having subtype WEEKDAY (so it can only have the values MON..FRI), X can be compared with SUN, even though SUN is not a permitted value of X.

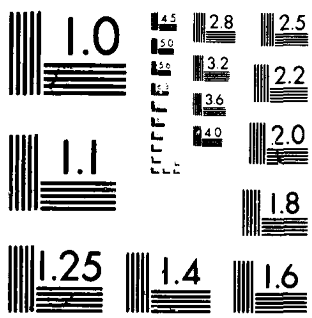
A predefined relational operation never raises an exception, although (of course), evaluation of an operand may cause an exception to be raised. A programmer-defined relational operator may, of course, raise an exception.

Detailed test objectives for enumeration types and predefined types are presented in separate subsections below.

AD-A091 760 SOFTECH INC WALTHAM MASS F/G 9/2
ADA COMPILER VALIDATION IMPLEMENTERS' GUIDE.(U)
OCT 80 J B GOODENOUGH MDA903-79-C-0687
UNCLASSIFIED 1067-2.3 NL

2 OF 4

AD
A091760



MICROCOPY RESOLUTION TEST CHART

NATIONAL BUREAU OF STANDARDS-1963-A

Compile-time Constraints

1. The operand types for relational and membership operators must be the same. (The operands need not have the same subtype, however.)

2. Ordering operations are predefined only for scalar and one-dimensional discrete array types, and hence, must be used only with these types unless user-defined definitions have been provided.

3. The membership operations cannot be overloaded (see 6.7).

4.5.2.a Enumeration Types

Test Objectives and Design Guidelines

1. Check that:

   - = and /= produce correct results, in particular, for operands having different subtypes;

   - the ordering of enumeration literals as defined by the ordering operators is the same as the order of occurrence of the literals in the type definition.

     Implementation Guideline: For example, check that

     (A < B) = (T'ORD(A) < T'ORD(B))

     holds for any A and B, where the subtypes of A and B may be different from each other and from T, and where T is a type common to both A and B.

   Implementation Guideline: Compare a variable of a subtype with a literal value not in the subtype's range.

2. Check the proper operation of the membership operators <u>in</u> and <u>not in</u>, using subtype names and explicit ranges as the second operand.

   Implementation Guideline: When a subtype name is used, at least one case should contain a first operand value that lies outside the range associated with the subtype name, e.g. MON <u>in</u> MIDWEEK;

3. Check that variables of two enumeration types having identical sets of enumeration values appearing in the same order cannot be compared with an equality or ordering operator. For example, given

```
type T is (A,B,C,D);
type U is (A,B,C,D);
X: T;
Y: U;
```

it is illegal to write X = Y or X < Y.

Check that the operands of a membership operation must all have the same type.

Implementation Guideline: Try literal values belonging to some other enumeration type as the first operand or as one of the range expressions. Also try variables of the wrong type in these positions.

4.5.2.b Character Types

Test Objectives and Design Guidelines

10. Check that an enumeration imposing an "unnatural" order on alphabetic characters, e.g., type T is ("C", "B", "E", "D"); yields the appropriate results for the ordering operators, e.g., T'("C") < T'("B").

4.5.2.c Boolean Types

Test Objectives and Design Guidelines

20. Check all combinations of the relational and membership operators for BOOLEAN values and ranges. Include the degenerate range FALSE..FALSE, the empty range TRUE..FALSE, and types derived from BOOLEAN.

4.5.2.d Integer Types

4.5.2.e Real Types

Semantic Ramifications

Note that the equality of model numbers is a consequence of the semantics of the type/subtype definitions. Hence this section is mainly concerned with values that are not model numbers. An implementation could be expected to provide more accurate semantics than the model requires. For details of the semantics, see section 4.5.8 of the LRM.

With

```
type REAL is digits D;
subtype R is REAL ...;
X : REAL
if X in subtype_indication then ...
```

there is a problem if the subtype_indication contains an accuracy constraint without an included range constraint, since there is no dynamic test for accuracy in Ada (indeed, it is meaningless). Since it would be irregular to exclude this case either syntactically or semantically, the LRM states that it is permitted (see Gaps), and consequently one can write

if X in REAL digits D then

even though the condition can be evaluated at compile-time and is always true.

Evaluation of a relational or membership operation must not raise NUMERIC_ERROR, even if, for example, the evaluation of "<" is performed by subtraction.

Compile-time Constraints

1. The two operands must have the same base type (after possible implicit conversion from universal real).

Exceptions

1. CONSTRAINT_ERROR is raised if a literal operand of a relational operator or membership test lies outside the range of the base type of the other operand (since implicit conversion is performed; see 3.5.6). (This check is suppressed by applying RANGE_CHECK to the base type or any of its subtypes.)

Test Objectives and Design Guidelines

1. For floating point types, check the following:

   a. A /= B same as not (A = B),

   b. A < B same as not (A >= B),

   c. A > B same as not (A <= B),

   d. adjacent model numbers give correct result,

   e. non-model numbers with distinct model intervals give correct results (see 4.5.8),

   f. when the intersection of two model intervals is a single model number (see 4.5.8), the correct result is given.

2. Check that literal values out of the range of an implemented floating point type raise CONSTRAINT_ERROR.

11. For fixed point types, check the following:

   a. A /= B same as not (A = B),

   b. A < B same as not (A >= B),

   c. A > B same as not (A <= B),

   d. adjacent model numbers give correct result,

e. non-model numbers with distinct model intervals give correct results (see 4.5.8),

f. when the intersection of two model intervals is a single model number (see 4.5.8), the correct result is given.

2. Check that literal values out of the range of an implemented fixed point type raise CONSTRAINT_ERROR.

Gaps

1. With the relational operators, the Brown model (as stated in 4.5.8) does not require consistency, i.e., not (A < B) being equal to A >= B etc. However, such inconsistencies could lead to severe problems and hence they are reported as errors by the tests above. Of course, the results are consistent for model numbers. The possibility of an inconsistency only arises when one of the operands is not a model number, but instead, lies between model numbers.

2. The statement in the LRM, "A test for an accuracy constraint [in a subtype indication] always yields the result TRUE," is intended to apply only for an accuracy constraint that does not include a range_constraint. This intention is certainly not indicated by the current wording. In addition, the current phrasing does not take into account the fact that the test

```
if X in R digits D+1 then
```

will raise CONSTRAINT_ERROR if R'DIGITS = D, since for the accuracy constraint to be compatible with R (see 3.5.7), D+1 must be less than or equal to R'DIGITS; an incompatible constraint in a subtype indication causes CONSTRAINT_ERROR to be raised (see 3.3).

We have assumed that the statement cited above applies only if CONSTRAINT_ERROR is not raised and only if the subtype indication of a membership test does not specify a range constraint. The definition of "satisfying a floating point constraint" in 3.5.7 should therefore be augmented accordingly, to specify what happens if the floating point constraint does not include a range constraint. Given this interpretation, the following anomalous cases can still arise:

```
type R is digits D;
subtype S is R digits D range 1.0 .. 10.0;
subtype T is R digits D;
subtype U is S digits D;
X: R := 11.0;
```

The test X in S yields FALSE, since the value of X is greater than S'LAST, the subtype indication does not contain an accuracy constraint, and S is declared with a range constraint. However, the test X in S digits S'DIGITS yields TRUE, since the accuracy constraint in the subtype indication does not contain a range constraint. Similarly, the test X in T yields TRUE since there is no range constraint in the definition of T or

R. The test X in S digits S'DIGITS range S'FIRST .. S'LAST yields FALSE, since the subtype indication does include a range constraint. It is unclear whether the test X in U yields TRUE or FALSE. Similar situations arise for fixed point types.

4.5.3 Adding Operators

4.5.3.a Boolean Adding Operators

4.5.3.b Integer Adding Operators

4.5.3.c Real Adding Operators

Semantic Ramifications

The semantics of real addition is a consequence of the semantics of all the operations giving a real result (see 4.5.8). Hence the testing required is determined by the relationship between the two operands, the result, and the corresponding model intervals. The model allows computations to be performed with an 'overlength accumulator', i.e., with more precision in the registers than the stored values (or the model numbers) demand.

A computer may have a single instruction to extend the precision of two operands and perform the double length operation. Hence, consider:

```
D1, D2, D3: DOUBLE;
A, B, C: SINGLE;
-- calculate A and B
C := A + B;
D1 := DOUBLE(A + B);
D2 := DOUBLE(A) + DOUBLE(B);
D3 := DOUBLE(C);
```

Does D1=D2? The answer is, not necessarily. A simple compiler will use single length addition for D1 and then convert to double length. However, it could use double length addition (which could be advantageous if this can be done in a single instruction) and hence get the same value as D2. Note that the code generated should be consistent. For instance, if the items are stored as the names suggest, then a compiler should not generate a double length fetch from a single length value and consequently pick up garbage from the second location. In principle, this is permitted by the Brown model, but non-determinism is hardly a desirably property. Hence one requires that D1=D2 or D1=D3, although the particular equality could depend upon the context.

The predefined floating point adding operators are derived from the declarations given in Appendix C (see 3.4 and 3.5.7), and hence, the operands must have the same base type. For fixed point, we assume this is also a requirement, although the LRM is not clear on this point (see Gaps). Note

that "+" and "-" can be overloaded by a user to accept operands of different base types.

Compile-time Constraints

1. The operands for predefined "+" and "-" must have the same base type for both fixed and floating point types.

Exceptions

1. The exception NUMERIC_ERROR is raised if the result is outside the implemented range, but only if MACHINE_OVERFLOWS (see 13.7.1) is true. Hence, when testing for exceptions, the attribute MACHINE_OVERFLOWS must be used (see Gaps). (This check is suppressed by applying OVERFLOW_CHECK to the operand type.)

Test Objectives and Design Guidelines

1. For floating point types, check that for the operator "+", correct results are produced when:

   a. A, B, and A+B are all model numbers.

   b. A and B are model numbers, but A+B is not.

   c. A is a model number but B and A+B are not.

   d. A, B and A+B are all model numbers with different subtypes.

   e. A and B are model numbers with different subtypes, but A+B is not a model number.

2. For floating point types, check that for the operator "-", correct results are produced when:

   a. A, B, and A-B are all model numbers.

   b. A and B are model numbers, but A-B is not.

   c. A is a model number but B and A-B are not.

   d. A, B and A-B are all model numbers with different subtypes.

   e. A and B are model numbers with different subtypes, but A-B is not a model number.

3. Check that NUMERIC_ERROR is raised unless MACHINE_OVERFLOWS is false for floating point types.

4. Check

   a. that non-associativity of real arithmetic is preserved, even when optimization would benefit from associativity of floating point,

b. subtype optimization (see 3.5.6.S) on addition with floating point,

c. subtype optimization on subtraction with floating point.

11. For fixed point types, check that for the operator "+", correct results are produced when:

a. A, B, and A+B are all model numbers.

b. A and B are model numbers, but A+B is not.

c. A is a model number but B and A+B are not.

d. A, B and A+B are all model numbers with different subtypes.

e. A and B are model numbers with different subtypes, but A+B is not a model number.

12. For fixed point types, check that for the operator "-", correct results are produced when:

a. A, B, and A-B are all model numbers.

b. A and B are model numbers, but A-B is not.

c. A is a model number but B and A-B are not.

d. A, B and A-B are all model numbers with different subtypes.

e. A and B are model numbers with different subtypes, but A-B is not a model number.

13. Check that NUMERIC_ERROR is raised unless MACHINE_OVERFLOWS is false for fixed point types.

14. Check

a. subtype optimization (see 3.5.6.S) on addition with fixed point,

b. subtype optimization on subtraction with fixed point.

<u>Gaps</u>

1. The LRM does not clearly state that the base type of two fixed point operands must be the same for predefined addition and subtraction.

2. 'MACHINE_OVERFLOWS is not defined for fixed point types (see 13.7.1). We assume this is an oversight.

4.5.3.d Array Adding Operators

4.5.4 Unary Operators

4.5.4.a Boolean Types

Compile-time Constraints

1. not is predefined only for Boolean types, which include types derived from the predefined BOOLEAN type.

Test Objectives and Design Guidelines

1. Check that not TRUE = FALSE, not FALSE = TRUE, and that a long sequence of nots, e.g., not (not (not (not A))) yields the appropriate value.

2. Check that non-Boolean arguments to not are forbidden, but that derived Boolean types are permitted.

4.5.4.b Integer Unary Operator

4.5.4.c Real Unary Operators

Semantic Ramifications

The unary plus operator has no semantic ramification except that +A is an expression whereas A may be a variable. The unary minus satisfies the same relationships as the other operations. Note that if A is a model number, then so is -A and hence the operation is exact.

Compile-time Constraints

1. The predefined unary "+" and "-" operators are defined only for integer, fixed, and floating point types.

Exceptions

1. Note that for a twos complement representation of floating point or fixed point, unary minus could raise NUMERIC_ERROR, but this is not possible with a model number. (This check is suppressed by applying OVERFLOW_CHECK to the operand type.)

Test Objectives and Design Guidelines

1. For floating point types, check that

   a. +A is equal to A and

b. -(-A)=A for model numbers.

11. For fixed point types, check that

a. +A is equal to A and

b. -(-A)=A for model numbers.

4.5.4.d Array Unary Operator

4.5.5 Multiplying Operators

4.5.5.a Integer Multiplying Operators

4.5.5.b Real Multiplying Operators

Semantic Ramifications

There are effectively four multiplication operations to check and three for division. The difficult cases for both are those for two fixed point operands. The reason for the difficulty is that the potential number of tests is cubed due to the need to consider the type of each operand and the result type separately.

The original Brown model considered division as a compound operation of reciprocation and multiplication. In Ada, this is one operation that could cause problems with some machines. It is conceivable that the precision of the machine as viewed by Ada (MANTISSA, etc.) may have to be reduced to allow for the inaccuracy of the division. The tests here should be adequate to ensure that MANTISSA has been set correctly. If a test fails, a numerical analyst should be consulted. The Cray 1 and Interdata 8/32 are thought to be machines which do not support division in the sense of Brown. Since it would be too expensive to use software division, some implementations for such machines may decide to deviate from the Standard (to a minor extent).

A correct implementation of fixed point is to store all values (including those in registers) as multiples of the ACTUAL_DELTA. However, if the ACTUAL_DELTA does not correspond to a convenient single/double word on the machine, then an implementation could choose not to mask off the unneeded bits in a register. Additional accuracy would be obtained with more efficient code. This is analogous to the extra precision often obtained with floating point. The source of extra bits are the operations fixed*fixed, fixed/integer, fixed/fixed and possibly constants. Of course, if the ACTUAL_DELTA value is not set by a representation specification, then the compiler could choose a value for it which corresponds to a whole word boundary. Again, subtype optimization (see 3.5.6.S) could result in unexpected inequalities in an analogous way to floating point.

An important special case with the operations fixed*fixed and fixed/fixed is when one operand is a literal (or literal expression). Since the precision of the literal is regarded as arbitrary large, the bounds on the result are determined by the other operand and the result type. If the literal is represented in the machine with as much precision as possible and with the same length as the other operand, then the maximum accuracy that can be guaranteed can be given by use of the usual integer division hardware.

Compile-time Constraints

1. If the predefined "*" and "/" operators are invoked with a floating point operand, both operands must be of the same base type.

2. The "mod" and "rem" operators are not predefined for floating point types.

3. If the predefined "*" operator is invoked for a fixed point type, one of the operands must be a fixed point type and the other must be of an integer or fixed point type.

4. If the predefined "/" operator is invoked for a fixed point type, the first operand must be of a fixed point type and the second must be of either an integer or fixed point type. (Note: the base types do not have to be the same.)

5. "mod" and "rem" are not predefined for fixed point types.

Exceptions

1. The exception NUMERIC_ERROR is raised if the result is outside the implemented range, but only if MACHINE_OVERFLOWS (see 13.7.1) is true. Hence, when testing for exceptions, the attribute MACHINE_OVERFLOWS must be used (see Gaps). (This check is suppressed by applying OVERFLOW_CHECK to an operand of a fixed or floating point type.)

2. NUMERIC_ERROR is raised if the value of the divisor is zero. (This check is suppressed by applying DIVIDE_CHECK to either of the operand types.)

Test Objectives and Design Guidelines

An interesting test for * is to generate 'LARGE by non-trivial factors. This requires that 2**MANTISSA-1 is not prime. If MANTISSA is composite, say N*M, then a factorization is available, namely:

2**MANTISSA-1 = (2**N-1)(2**(N*(M-1)) + 2**(N*(M-2)) + ... + 1)

Hence this number is only prime if MANTISSA is prime, in which case, 'LARGE is then a Mersenne prime. The following values of MANTISSA give rise to Mersenne primes: 2, 3, 5, 7, 13, 17, 19, 31, 61, 89, 107, 127, 521, 607. (See Knuth Vol 2, p356). The factorizations used in the test programs are as follows:

N	2**N - 1	=	P	*	Q

5	31	=	31 *	1
6	63	=	9 *	7
7	127	=	127 *	1
8	255	=	17 *	15
9	511	=	73 *	7
10	1023	=	33 *	31
11	2047	=	89 *	23
12	4095	=	91 *	45
13	8191	=	8191 *	1
14	16 383	=	129 *	127
15	32 767	=	217 *	151
16	65 535	=	257 *	255
17	131 071	=	131 071 *	1
18	262 143	=	513 *	511
19	524 287	=	524 287 *	1
20	1 048 575	=	1025 *	1023
21	2 097 151	=	2359 *	889
22	4 194 303	=	2049 *	2047
23	8 388 607	=	178 481 *	47
24	16 777 215	=	4097 *	4095
25	33 554 431	=	1 082 401 *	31
26	67 108 863	=	8193 *	8191
27	134 217 727	=	262 657 *	511
28	268 435 455	=	16 385 *	16 383
29	536 870 911	=	2 304 167 *	233
30	1 073 741 823	=	32 769 *	32 767
31	2 147 483 647	=	prime	
32	4 294 967 295	=	65 537 *	65 535
33	8 589 934 591	=	599 479 *	14 329
34	17 179 869 183	=	131 073 *	131 071
35	34 359 738 367	=	8 727 391 *	3937
36	68 719 476 735	=	262 145 *	262 143
37	137 438 953 471	=	616 318 177 *	223
38	274 877 906 943	=	524 289 *	524 287
39	549 755 813 887	=	9 588 151 *	57 337
40	1 099 511 627 775	=	1 048 577 *	1 048 575
41	2 199 023 255 551	=	factors > 1723	
42	4 398 046 511 103	=	2 097 153 *	2 097 151
43	8 796 093 022 207	=	20 408 568 497 *	431
44	17 592 186 044 415	=	4 194 305 *	4 194 303
45	35 184 372 088 831	=	1 073 774 593 *	32 767
46	70 368 744 177 663	=	8 388 609 *	8 388 607
47	140 737 488 355 327	=	factors > 471	
48	281 474 976 710 655	=	16 777 217 *	16 777 215

```
49       562 949 953 421 311  =  4 432 676 798 593 *          127

50     1 125 899 906 842 623  =        33 554 433 *  33 554 431
51     2 251 799 813 685 247  =    17 180 000 257 *     131 071
52     4 503 599 627 370 495  =        67 108 865 *  67 108 863
53     9 007 199 254 740 991  =
54    18 014 398 509 481 983  =       134 217 729 * 134 217 727

55    36 028 [illegible] 018 963 967  = 17 600 780 175 361 *       2047
56    72 057 594 037 927 935  =       268 435 457 * 268 435 455
57   144 115 188 075 855 871  =
58   288 230 376 151 711 743  =       536 870 913 * 536 870 911
59   576 460 752 303 423 487  =

60     1 152 921 504 606 846 975 =      1 073 741 825 * 1 073 741
823
61     2 305 843 009 213 693 951 =   prime
62     4 611 686 018 427 387 903 =      2 147 483 649 * 2 147 483
647
63     9 223 372 036 854 775 807 =  20 303 320 287 433 *       454
279
64    18 446 744 073 709 [illegible] 615 =      4 294 967 297 * 4 294 967
295

65    36 893 488 147 419 103 231 = 145 295 143 558 111 *       253
921
66    73 786 976 294 838 206 463 = 572 165 044 430 783 *       128
961
67   147 573 952 589 676 412 927 = factors > 1609
68   295 147 905 179 352 825 855 =   17 179 869 185 *   17 179 869
183
69   590 295 810 358 705 651 711 =   10 052 678 938 039 *   58 720
249

70     1 180 591 620 717 411 303 423 =          34 359 738 369
                                     *          34 359 738 367
71     2 361 183 241 434 822 606 847 =
72     4 722 366 482 869 645 213 695 =          68 719 476 737
                                     *          68 719 476 735
73     9 444 732 965 739 290 427 391 =
74    18 889 465 931 478 580 854 783 =         137 438 953 473
                                     *         137 438 953 471

75    37 778 931 862 957 161 709 567 =     160 842 848 628 151
                                     *             234 881 017
76    75 557 863 725 914 323 419 135 =         274 877 906 945
                                     *         274 877 906 943
77   151 115 727 451 828 646 838 271 = 581 283 643 249 112 959
                                     *                 259 969
78   302 231 454 903 657 293 676 543 =         549 755 813 889
                                     *         549 755 813 887
```

```
79   604 462 909 807 314 587 353 087 =

80     1 208 925 819 614 629 174 706 175 =          1 099 511 627 777
                                         *          1 099 511 627 775
81     2 417 851 639 229 258 349 412 351 =     18 014 398 643 699 713
                                         *                134 217 727
82     4 835 703 278 458 516 698 824 703 =          2 199 023 255 553
                                         *          2 199 023 255 551
83     9 671 406 556 917 033 397 649 407 =
84    19 342 813 113 834 066 795 298 815 =          4 398 046 511 105
                                         *          4 398 046 511 103
85    38 685 626 227 668 133 590 597 631 =  9 520 972 806 333 758 431
                                         *                  4 063 201
86    77 371 252 455 336 267 181 195 263 =          8 796 093 022 209
                                         *          8 796 093 022 207
87   154 742 504 910 672 534 362 390 527 =        417 576 898 368 951
                                         *                3 758 096 377
88   309 485 009 821 345 068 724 781 055 =         17 592 186 044 417
                                         *         17 592 186 044 415
89   618 970 019 642 690 137 449 562 111 = prime

90     1 237 940 039 285 380 274 899 124 223 =
                                                   35 184 372 088 833
                                             *     35 184 372 088 831
91     2 475 880 078 570 760 549 798 248 447 =
                                       2 380 065 770 834 284 748 671
                                             *                1 040 257
92     4 951 760 157 141 521 099 596 496 895 =
                                                   70 368 744 177 665
                                             *     70 368 744 177 663
93     9 903 520 314 283 042 199 192 993 791 =
                                              658 812 288 653 553 079
                                             *         15 032 385 529
94    19 807 040 628 566 084 398 385 987 583 =
                                                  140 737 488 355 329
                                             *    140 737 488 355 329

95    39 614 081 257 132 168 796 771 975 167 =
                                       2 437 355 091 657 331 538 911
                                             *               16 252 897
96    79 228 162 514 264 337 593 543 950 335 =
                                                  281 474 976 710 657
                                             *    281 474 976 710 655
97   158 456 325 028 528 675 186 087 900 671 =
98   316 912 650 057 057 350 372 175 801 343 =
                                                  562 949 953 421 313
                                             *    562 949 953 421 311
99   633 825 300 114 114 700 744 351 602 687 =
                                            73 786 976 303 428 141 057
                                             *            8 589 934 591
```

```
100    1 267 650 600 228 229 401 488 703 205 375 =
                                         1 125 899 906 842 623
                                       * 1 125 899 906 842 621
```

1. Check that for the predefined floating point "*" operator, correct results are produced when:

   a. A, B, and A*B are all model numbers.

   b. A and B are model numbers, but A*B is not.

   c. A is a model number, but B and A*B are not.

   d. A, B and A*B are all model numbers with different subtypes.

   e. A and B are model numbers with different subtypes, but A*B is not a model number.

2. Check that for the predefined floating point "/" operator, correct results are produced when:

   a. A, B, and A/B are all model numbers.

   b. A and B are model numbers, but A/B is not.

   c. A is a model number, but B and A/B are not.

   d. A, B and A/B are all model numbers with different subtypes.

   e. A and B are model numbers with different subtypes, but A/B is not a model number.

3. Check that (a) the same floating point type must be used with a multiplying operators, (b) "mod" and "rem" cannot be used with floating point operands.

11. Check the following for a variety of fixed point types:

   a. fixed*integer when all values are model numbers.

   b. fixed*integer values are bounded correctly for non-model numbers (check same values with integer*fixed).

   c. fixed/integer when all values are model numbers.

   d. fixed/integer values are bounded correctly for non-model numbers.

12. Special tests are needed for fixed*fixed and fixed/fixed because of the three types involved. Check for a variety of of the three types involved:

   a. fixed*fixed with all values being model numbers.

b. fixed*fixed with operands as model numbers but results are not.

c. fixed*fixed with no model numbers.

d. fixed/fixed with all values being model numbers.

e. fixed/fixed with operands as model numbers but results are not,

f. fixed/fixed with no model numbers.

13. Check that "mod" and "rem" cannot be used with fixed point operands.

Gaps

1. The LRM does not state explicitly that for the fixed point operations involving integers, the 'model numbers' of the integers are the integers.

4.5.6 Exponentiating Operator

4.5.6.a Integer Exponentiating Operator

4.5.6.b Real Exponentiating Operator

Semantic Ramifications

Since the operation is defined in terms of repeated multiplication, no specific additional semantics is needed. Note that since 0.0**0 =1.0, it may be necessary to check explicitly for this case if, for instance, overflow is to be avoided.

Exceptions

1. For floating point exponentiation, the only possible exception is NUMERIC_ERROR which arises in the same manner as for the multiplying operators. (This check can be suppressed by applying OVERFLOW_CHECK to the floating point type or to the integer type.)

Test Objectives and Design Guidelines

1. Check that ** is implemented for a variety of floating point and integer types and

   a. X ** 0 = 1.0, X ** 1 = X, and X ** (-1) = 1.0/X.

   b. Error bounds on the numerical result are acceptable.

   c. Large variable exponent values are accepted.

2. Check that NUMERIC_ERROR is raised unless MACHINE_OVERFLOWS is false.

11. Check that fixed**integer is not permitted.

Gaps

1. The phrase "equivalent to repeated multiplication" is potentially ambiguous. An implementation is presumably free to perform exponentiation with fewer multiplications (see Knuth Vol 2). Note that although (X*X)*(X*) /= ((X*X)*X)*X, both expressions have similar error bounds with classical error analysis. With the Brown model, the two are distinguishable, but the intention of the above phrase is presumably to permit any form of repeated multiplication.

4.5.7 The Function ABS

4.5.7.a The Integer Function ABS

4.5.7.b The Real Function ABS

Semantic Ramifications

The ABS function is available for all numeric types - integer, fixed point, and floating point. For model numbers, the exact result is obtained since model values are symmetric with respect to the sign. With a fixed point type F filling a word on a twos complement machine, the expression -F'FIRST will overflow and hence should raise the NUMERIC_ERROR exception. However, if a spare bit is kept at the top of the word to allow comparison by subtraction, the expression will not overflow but -F'FIRST is still larger than 'LAST and hence CONSTRAINT_ERROR is raised on assignment.

Compile-time Constraints

1. The predefined ABS function is only defined for arguments of a numeric type.

Exceptions

1. NUMERIC_ERROR is raised when the result exceeds the implemented range of the operand's base type. It may also be raised when the model interval of the result overflows (see 4.5.8). On most machines, ABS cannot raise overflow with floating point, and only on one value with fixed point (and integer). (This check can be suppressed by applying OVERFLOW_CHECK to the operand type.)

Test Objectives and Design Guidelines

1. For a variety of floating point types, check:

   a. for model number A >= 0, that ABS(A)=A,

b. for model numbers A <= 0, that ABS(A)=-A,

c. for values that are not model numbers, values are within the appropriate model interval.

11. For a variety of fixed point types, check:

a. for model number A >= 0, that ABS(A)=A,

b. for model numbers A <= 0, that ABS(A)=-A,

c. for values that are not model numbers, values are within the appropriate model interval.

4.5.8 Accuracy of Operations with Real Operands

The semantics in this section is taken into account in the testing of the appropriate operators. See sections 4.5.2 - 4.5.7 and 4.6.

4.6 Type Conversion

4.6.a Enumeration Type Conversion

4.6.b Integer Type Conversion

4.6.c Real Type Conversions

Semantic Ramifications

The semantics is the same as for the other operations. A value is widened to a model interval of the type or subtype of the expression. The resulting interval is converted mathematically and again widened to a model interval of the resulting type. The last interval bounds the values permitted by the conversion.

Exceptions

1. The exception NUMERIC_ERROR can only be raised (but need not) if one of the intervals overflows. (This check can be suppressed by applying OVERFLOW_CHECK to the type of the conversion result.)

2. If T is the type_mark used in the type_conversion and T is a scalar type, CONSTRAINT_ERROR is raised if the value of the converted result does not lie in the range T'FIRST..T'LAST. (This check is suppressed by applying RANGE_CHECK to T.)

Test Objectives and Design Guidelines

Specific tests are needed between the pairs of numeric types - integer, floating point and fixed point. The cases to be handled are exact conversion with model numbers and other values.

1. Floating point to integer: test rounding.

2. Fixed point to integer: test rounding.

3. Floating point to fixed point.

4. Integer to fixed point.

5. Fixed point to fixed point.

6. Floating point to floating point.

7. Fixed point to floating point.

8. Integer to floating point.

Gaps

1. The current wording of the LRM does not make it clear that a type conversion is an operation giving a real result in the sense of section 4.5.8.

4.8 Allocators

Semantic Ramifications

The type matching requirements of the various contexts in which an allocator can be used determines the type of the value returned by the allocator:

- in assignment statements, the type must be the type of the left hand side;

- in equality comparisons, the type must be the type of the other operand;

- in component and object declarations, the type of the initial value must be the type of the declared component or object, respectively.

- in an aggregate, the type must be the component type required by the aggregate's type;

- as a default value in a parameter list, the type must be the type of the formal parameter;

- in a return statement, the type must be the type of the value to be returned;

- as an actual in parameter, the type must be that of the corresponding formal parameter.

Ambiguities in the type of an allocator can only arise if the type of the allocated object appears in more than one access type definition, e.g.,

```
type PEOPLE is access PERSON;
type HUMAN  is access PERSON;
```

The allocator, new PERSON (...), can be either of type PEOPLE or HUMAN. Since in either case, an object of type PERSON is allocated, what are the implications of this type determination? There are two possibilities:

- if a collection size has been specified for PEOPLE that is smaller than the collection size for HUMAN (see 13.2) a PEOPLE allocation could raise STORAGE_ERROR when a HUMAN allocation would not;

- if PEOPLE values are represented with offset pointers (see 13.2) and HUMAN values with full pointers, then the correct interpretation of the run-time value depends on correctly determining its type.

And of course, knowing the possible types of new PERSON (...) is important in overloading resolution. Suppose we have

```
procedure F(X: HUMAN);
procedure F(X: DATA);
```

and DATA is not an access PERSON type. Then the call

```
F(new PERSON(...));
```

is clearly a call of the first procedure. It should also be noted that the expression:

```
new PERSON(...) = new PERSON (...)
```

would be illegal, since both operands have the same set of potential types (namely, PEOPLE and HUMAN). Even though the two allocators cannot possibly have the same value, a compiler cannot give the value FALSE for the above expression. The expression must be rejected as illegal because the types are ambiguous.

Ada is designed so that for a record or array type, an allocator need only allocate enough space to hold the particular subtype being allocated since no subsequent assignments are permitted to change the index or discriminant constraints associated with such an allocated object.

Note there is no requirement that the values of an index or discriminant constraint be static.

Note that for an access subtype, e.g.,

```
subtype MALE is PEOPLE(M);
```

an allocator new MALE is equivalent to new PEOPLE(M); (see LRM 3.3); moreover, new MALE(M) would be illegal, since a subtype name cannot be further constrained.

Note that a discriminant constraint and an aggregate using named notation can have the same syntactic form. The interpretation of such a construct depends on the type mark used in the aggregate. For example,

```
type INDEX is (A, B, C);
type ARR is array (A..C) of INTEGER;
type REC(A,C,B: INTEGER) is
   record ... end record;
type AGG(B, A: INTEGER) is
   record
       C: INTEGER;
   end record;
type ARR_NAME is access ARR;
type REC_NAME is access REC;
type AGG_NAME is access AGG;
...

new ARR (A => 3, B => 55, C => 10)      -- array aggregate
new REC (A => 3, B => 55, C => 10)      -- discriminant constraint
new AGG (A => 3, B => 55, C => 10)      -- record aggregate
new AGG (3, 55)           -- only two values, therefore, a discriminant
                          --    constraint
new AGG (3, 55, 10)       -- can't be a discriminant constraint for this
                          --    type
new REC (3, 55, 10)       -- is a discriminant constraint for this type
```

From an implementer's point of view, given new T X, where X has the form of an aggregate, one can proceed as follows:

- if T is a record type, see if X can serve as a discriminant constraint for T;
- if not, then see if X is a value of type T;
- if not, the allocator is illegal.

It should be noted that if T is a private type, X can only serve as a discriminant constraint, since there are no aggregates giving values of a private type. In addition, if X is a parenthesized single value, it can't be an aggregate, since one-component aggregates must use named notation (see 4.3).

If a type T contains no components except a discriminant, then new T(5) would be a legal allocator and the (5) must be a discriminant constraint,

since one-component aggregates cannot be given in positional notation. If ST is a subtype of T, e.g.,

```
subtype ST is T(5);
```

then ST(5) is illegal since (5) cannot be a discriminant constraint (constrained record types cannot be further constrained) and it cannot be an aggregate, since one-component aggregates must use named notation.

In general, the syntactic and semantic legality constraints for allocators are a combination of the constraints for declarative and assignment contexts, e.g., given new T X, the combination T X must either satisfy the constraints for an object declaration of the form

```
A: T X;
```

or given a variable of type T, e.g., V, the following statement must be legal:

```
V := X;
```

Moreover, depending on which of these contexts defines the legality of the allocator, the exception conditions that would be raised in that context are raised for the allocator as well.

For purposes of testing all the legal combinations of allocators it is helpful to use the following table, which specifies various potentially legal combinations of syntactic elements and explains what interpretations are potentially legal. The examples have been chosen to maximize the opportunity for ambiguity.

The column headings have the following meanings:

SS -- scalar subtype
UR -- unconstrained record type with at least one discriminant
CR -- constrained record type
UP -- unconstrained private type
CP -- constrained private type
ULP -- unconstrained limited private type
CLP -- constrained limited private type
UA -- unconstrained array type
CA -- constrained array type

The row headings have the following meanings:

(E) -- E is any scalar, record, or array expression not enclosed in parentheses

((E)) -- E is any scalar, record, or array expression not enclosed in parentheses

(1,2) -- a sequence of two or more components of the same type, enclosed in parentheses

((1,2)) -- a sequence of two or more components of the same type enclosed in two sets of parentheses

(others) -- a positional aggregate containing an others choice, e.g., (1,2,others => 0)

((others)) -- a positional aggregate enclosed in parentheses and containing an others choice

NULL -- X does not exist

The entries in the table refer to the notes following the table. An X indicates an illegal construct. 1, 2, and 3 refer to the syntactic alternatives for an allocator. The upper left entry represents an allocator having the form new SS(E).

	SS	UR	CR	UP	CP	ULP	CLP	UA	CA
(E)	1a	1,3	1d	1,3	1d	3b	x7	1e	1e
((E))	1a	1,3	1d	1,3	1d	3b	x7	1e	1e
(1,2)	x1	2,3	2a	3a	x2	3b	x7	2b	2b
((1,2))	x1	1b	1b	x3	x3	x3	x7	1b	1b
(others)	x1	2c	2c	x3	x3	x3	x7	x4	2d
((others))	x1	2c	2c	x3	x3	x3	x7	x4	2e
NULL	1c	x5	1c	x5	1c	x5	1c	x6	1c

1a E is the value of the allocated object

1b ((1,2)) is a parenthesized aggregate which must provide values for all the record or array components; note that ((1,2)) does not satisfy the syntax for discriminant constraints.

1c Initial values are not required for constrained types

1d E must be an expression yielding a value of type T and whose constraints equal T's constraints

1e E must yield an array value of type T with index constraints compatible with T's index type and/or constraints

1,3 (E) or ((E)) is a discriminant constraint if E is an expression yielding a scalar value; otherwise E must be an expression yielding a value of the record or private type.

2,3 (1,2) can be considered a discriminant constraint if the

record type has exactly two discriminants; otherwise, it must be an aggregate providing values for all discriminants and components of the record

2a (1,2) cannot be a discriminant constraint, since the record is already constrained; so it must be an aggregate providing values for all components of the record (see also 1d)

2b (1,2) is an array aggregate;

2c (<u>others</u>) must be a record aggregate, since it does not satisfy the syntax for a discriminant

2d (<u>others</u>) is an array aggregate

2e 4.3.2(c) suggests this usage is legal

3a (1,2) must be a discriminant constraint, since a complete value cannot be specified as an aggregate for a private type

3b the only values that can be provided for limited private types are discriminant constraints. Therefore, (E), ((E)), or (1,2) must be interpreted as discriminant constraints.

x1 An aggregate is not compatible with a scalar type

x2 (1,2) cannot be a discriminant constraint, since CP is aleady constrained; it can't be an aggregate, since aggregates can't be values of private types

x3 ((1,2)) and (<u>others</u>) does not satisfy the syntax for a discriminant constraint; it can't be an aggregate either, since aggregates can't be specified as values of private types

x4 An aggregate with <u>others</u> can't specified for an unconstrained array type (see 4.3.2)

x5 Values must be provided for discriminants even if the type declaration provides default discriminant values.

x6 Values must be provided for the bounds of an array type.

x7 Since the type is constrained no discriminant constraint can be provided; since the type is limited private, no initial value can be provided.

Note that an array aggregate of the form ((1,2,3), (1,2)) raises CONSTRAINT_ERROR, since it does not have the same number of components for each dimension.

Note that for an unconstrained array type, CONSTRAINT_ERROR can be raised

if there are too many components for the range associated with the index type. For example,

```
type TWO is INTEGER range 1..2;
type T is array (TWO range <>, TWO range <>) of INTEGER;
...
new T ((1,2,3), (1,2,3))        -- CONSTRAINT_ERROR
```

The allocator raises CONSTRAINT_ERROR since the type T requires a maximum of two components per dimension. Note that the bounds of TWO could have been specified with variables, so the appropriateness of the allocator cannot necessarily be determined at compile time. Similarly, an aggregate using named notation with index values outside the range specified for the index type TWO would raise CONSTRAINT_ERROR:

```
new T ((3|4 => 0), (1|2 => 0))  -- CONSTRAINT_ERROR
```

Compile-time Constraints

Detailed constraints are specified below for the various forms of allocators. All the constraints, however, derive from the following general rule:

> An allocator of the form new T X is legal if and only if T X is a legal subtype_indication or X is a value of type T.

1. An allocator of the form new T is permitted only if:

   - T is a constrained private, limited private, record, or array subtype; or
   - T is a private type, limited private type, or record type without discriminants; or
   - T is a scalar or access subtype;

2. An allocator of the form new T (V), where V is a scalar expression possibly enclosed in one or more levels of parentheses, is permitted only if:

   - T is a scalar subtype and the base type of V and T are the same; or
   - T is an unconstrained record type, private type, or limited private type with exactly one discriminant, and the base type of V and the base type of the discriminant are the same.

3. An allocator of the form new T (V), where V is a non-scalar expression other than an aggregate, is permitted only if T is an access, record, array, or private type and the base type of T and V are the same. (Note that T can be a constrained or unconstrained type.)

4. An allocator of the form new T A, where A has the form of a discriminant

constraint (i.e., does not contain forms such as <u>others</u> => D or a discrete_range) is permitted only if:

- T is an array or record type (constrained or unconstrained) and A is an aggregate of type T (see 4.3); or

- T is an unconstrained private or record type with more than one discriminant and A is a suitable discriminant constraint for T (see 3.7.2).

5. An allocator of the form <u>new</u> T A, where A is an aggregate containing no <u>others</u> choice, is permitted only if

   - T is an array or record type (constrained or unconstrained) and A is an aggregate of type T (see 4.3).

6. An allocator of the form <u>new</u> T (A), where A is an aggregate having no <u>others</u> choice and enclosed in zero or more sets of parentheses, is permitted only if

   - T is an array or record type (constrained or unconstrained) and A is an aggregate of type T (see 4.3);

7. An allocator of the form <u>new</u> T A0, where A0 is an aggregate containing an <u>others</u> choice, is permitted only if:

   - T is a constrained or unconstrained record type and A0 is an aggregate of type T (see 4.3.1); or

   - T is a constrained array type and A is an aggregate of type T (see 4.3.2);

8. An allocator of the form <u>new</u> T (A0), where A0 is an aggregate containing an <u>others</u> choice but no other choices and where A0 is enclosed in zero or more sets of parentheses, is permitted only if:

   - T is a constrained or unconstrained record type and A0 is an aggregate of type T; or

   - T is a constrained array type and A0 is an aggregate of type T.

9. An allocator may contain an index_constraint if and only if T, the type_mark in the allocator, denotes an unconstrained array type and the index constraint is suitable for T (see 3.6.1).

10. For an allocator of the form <u>new</u> T X, where T is a multidimensional array type (constrained or unconstrained) and X is an array aggregate, the number of the components for a given dimension must be the same for each element of that dimension.

Exceptions

1. For an allocator of the form new T X, CONSTRAINT_ERROR is raised if:

   - T is a scalar type and X does not satisfy T's range constraint.

   - T is an unconstrained access, record, private, or limited private type, X is a discriminant constraint, and at least one of the values of X is outside the range of the corresponding discriminant.

   - T is a constrained record type and

     - at least one of the discriminant values in X does not equal the corresponding discriminant value of T; or

     - at least one of the non-discriminant components of X fails to satisfy the subtype constraint of the corresponding component of T.

   - T is a constrained private type and at least one of the discriminant values in X does not equal the corresponding discriminant value of T.

   - T is an unconstrained array type and

     - at least one of the components of X fails to satisfy the subtype constraint of T's component type; or

     - if X is an aggregate using named notation, at least one of the choice values lies outside the range of values permitted for the corresponding index type; or

     - if X is a multi-dimensional aggregate, there are more components for a given index than are permitted for the corresponding index type in T.

   - T is a constrained array type and

     - at least one of the components of X fails to satisfy the subtype constraint of T's component type; or

     - if X is an aggregate using named notation, at least one of the choice values lies outside the range of values for the corresponding index subtype; or

     - the number of components for a given dimension does not satisfy the index constraint for the corresponding index of T.

2. STORAGE_ERROR is raised if insufficient storage remains for allocating objects of the specified type.

Test Objectives and Design Guidelines

1. Check that illegal forms of allocators are forbidden. In particular, for allocators of the form new T, check that:

   - T cannot be an unconstrained record, private, or limited private type, even one with default discriminant values;

   - T cannot be an unconstrained array type.

2. Check that illegal forms of allocators are forbidden, in particular, for allocators of the form new T X, check that:

   - if T is a scalar type, X cannot be a positional aggregate, or have the form (), or be an aggregate containing a single others choice;

   - if T is a record, private, or limited private type with d discriminants and c components (for a given set of discriminant values), and if X contains x values, then:

     - if T is an unconstrained record type and X has the form of a positional or named aggregate,

       - x cannot be less than d;

       - x cannot be greater than d and less than d+c;

       - x cannot be greater than d+c;

     - if T is an unconstrained private or unconstrained limited private type and X has the form of a positional aggregate,

       - x cannot be less than d;

       - x cannot be greater than d;

       - x cannot equal the number of components in the underlying private type definition;

     - if T is unconstrained, X cannot have the form of a parenthesized aggregate containing d values;

       Implementation Guideline: Except for the extra set of parentheses, X should be a suitable discriminant constraint for T.

   - if T is a private or limited private type (constrained or unconstrained), X cannot be a parenthesized or unparenthesized aggregate with an others choice;

   - if T is unconstrained with two integer discriminants named A and B, X cannot have the form (A => 0, B => 0, A => 0) (see 3.7.2);

- if T is a constrained record type (possibly with no components) and X is of type T, let X contain a constraint value not equal to the corresponding constraint value for T. Use both named and positional notation.

- if T is a constrained private or limited private type, let X have the form of a positional aggregate whose values all equal the corresponding constraint values; also let X have the form of a complete and valid aggregate for the underlying private type definition.

- if T is an unconstrained array type, X cannot be a parenthesized or unparenthesized aggregate with an others choice;

- if T is a limited private type, X cannot be a parenthesized value of type T.

3. For allocators of the form new T X, check that CONSTRAINT_ERROR is raised if:

   - T is a scalar subtype and X is outside the range of T;

   - T is an unconstrained record or private type, X is a discriminant constraint, and one of the values of X is outside the range of the corresponding discriminant;

   - T is a constrained record type, X is an aggregate, and one of the discriminant values in X does not equal the corresponding constraint value or one of the non-discriminant values is:

     - an out-of-range scalar value;

     - an aggregate with too many or too few values (the number of values should be: independent of any discriminant value; dependent on a static discriminant value; dependent on a non-static discriminant value);

   - T is an unconstrained array type and one of the values of X is:

     - an out-of-range scalar value;

     - an aggregate with too many or too few values (the required number of values should be both statically and non-statically defined);

   - T is a constrained array type and X contains too many or too few values (both static and non-static constraints should be tried);

   - X is an index constraint whose range for one index exceeds the range of the corresponding index subtype;

   - T is a constrained or unconstrained multi-dimensional array type and all components of X do not have the same length.

4. Check that the form new T is permitted if

   - T is a scalar subtype; or
   - T is a constrained record, private, limited private, or array type; or
   - T is a record, private, or limited private type without constraints.

   For the cases where T is a record type, check that any components given default values in the type definition have the appropriate default values after the execution of the allocator, and check that the constraints have the appropriate values.

5. Check that an allocator of the form new T X allocates a new object each time it is executed and that:

   - if X is an expression and T a scalar type, the allocated object has the value of X;
   - if X is an aggregate and T is a record or array type (constrained or unconstrained), the allocated object has the value of X;
   - if X is a discriminant constraint and T is an unconstrained record or private type, the allocated object has the discriminant value specified by X;
   - if X is an index constraint and T an unconstrained array type, the allocated object has the index bounds specified by X.

   Implementation Guideline: X should contain all static values in some tests and non-static values in others, e.g., one component of X might be a loop parameter.

4.9 Static Expressions

4.9.a Enumeration Static Expressions

4.9.b Integer Static Expressions

4.9.c Real Static Expressions

Semantic Ramifications

Real static expressions appear in real type definitions (see 3.5.7 and 3.5.9) and in the representation specification for ACTUAL_DELTA (see 13.2). These expressions must be evaluated by the compiler (at least as far as the

effect on the program is concerned). Since type conversions can appear in static expressions such as INTEGER(<some real expression>), if the implementation requires any value at compile-time, this includes real expressions.

Real static expressions will be evaluated by the host rather than the target computer, and so will be evaluated with potentially radically different underlying hardware. However, such expressions are typed with a known accuracy, and both the target and host must evaluate to at least that accuracy. A host would presumably use the same computational package for such evaluations as for literal expressions (since literal expressions must be evaluated with high accuracy, see 4.10). Hence host evaluation is likely to be more accurate and more efficient (at run-time).

A compiler which has the ability to evaluate real static expressions is likely to use this package to optimize ordinary expressions (expressions which happen to be static). This is safe, but substantial care is necessary. For instance, if X is a floating point variable, then X + 0.5 - 0.5 cannot be optimized to X because the parsing is (X+0.5)-0.5 giving no static subexpression. Reordering is not permitted since floating point addition is not associative. (Of course, if X + (0.5 - 0.5) had been written, the (0.5-0.5) can be replaced by 0.0 since it is a literal expression). Another difficulty in the evaluation of real static expressions is that of ensuring that the necessary exceptions are still raised at run time (see 10.6).

Compile-time Constraints

The eight conditions must be checked by the compiler. The compiler must reject a program which contains more than the permitted constituents in an expression, e.g., a function call such as SQRT(2.0). An aggregate cannot appear in a real static expression (case (b)).

Exceptions

The exception conditions are the same as for dynamic evaluation. However, host evaluation would typically involve more accuracy (and exponent range) than the target and hence different results would be obtained for values out of range of the model.

Test Objectives and Design Guidelines

Checks for specific constituents that are not one of the eight permitted classes are not included here. The tests specified here are just those involving real types.

1. Check that each of the seven permitted constituents are allowed:

   a. literal expression as static expression,

   c. constant initialized by a static expression,

   d. predefined operators and ABS,

e. static attribute in static expression,

f. type conversion,

g. selected component,

h. indexed component.

4.10 Literal Expressions

4.10.a Integer Literal Expressions

4.10.b Real Literal Expressions

Semantic Ramifications

A number declaration serves to give a literal value an identifier. A literal value may be an exact (integer) literal, or an approximate (real) literal. Since the corresponding literal value may have to be computed from an expression, number declarations imply compile-time evaluation of expressions. It is important that such expressions are handled correctly.

An approximate real expression can use integer literals as well as real literals, because an integer multiple of a real is a real; real literal expressions share the capability of floating point and fixed point expressions. Clearly, an implementation must allow any numerical value to be set with the maximum accuracy provided by the types permitted. For both floating point and fixed point, an implementation is likely to set upper limits on the total accuracy. Real literals must be stored by the compiler with a little more precision than this (say 10 bits more) in order to allow for rounding errors in the evaluation of literal expressions and to ensure correct rounding when necessary (see 4.2). Since model numbers are binary for floating point and usually binary for fixed point, if the compiler stores number declaration values internally in decimal, it is essential that values are rounded when converted to an ordinary type (otherwise model numbers will not be stored exactly).

Given a large literal value, say

```
BIG_L: constant := 1_000_000_000;
```

then

```
I: INTEGER := BIG_L;
```

will raise CONSTRAINT_ERROR since it is outside the range of INTEGER. The exception is caused by the implicit conversion from Universal Integer to INTEGER (see 3.5.4). On the other hand:

```
I: INTEGER := BIG_L - BIG_L;
```

will initialize I to zero since the expression (BIG_L - BIG_L) is a literal expression that is converted to INTEGER after evaluation.

Compile-time Constraints

An exact expression must only use integer literals and names denoting integer literals.

Exceptions

The only exceptions that can arise in the evaluation of a number declaration are due to division by zero and that corresponding to the range constraint on the exponentiation operator.

Test Objectives and Design Guidelines

1. Each integer operation with various signs for operands.

2. Large integer values with small resulting number declaration values by means of subtracting two large values.

3. Use of other number declarations.

4. Large integer values and resulting number declaration values.

5. Exception - integer division by zero.

6. Exception - exponentiate of integer with negative exponent.

7. Integer values near MAX_INT.

8. Check the use of an integer literal expression in

   a. an expression,

   b. a choice in a case statement,

   c. for conditional compilation.

11. Each real operation with various signs for operands.

12. Large real value with small resulting number declaration value by means of subtracting two large values (cancellation)

13. Use of other real and integer number declarations.

14. Large real values and resulting number declaration values.

15. Exception - real division by zero.

16. Compile-time constraint violation - real + integer, and real ** real.

17. Check round-off in relation to MAX_DIGITS and MIN_DELTA.

18. Check that a literal value given in a number declaration is bounded by appropriate model intervals for various real types.

Gaps

1. The LRM does not state if values greater than MAX_INT can be used in number declarations. An implementation is likely to place some limit, but this should be very large, at least for integers. Consider the likely use of conditional compilation in Ada to generate code for different machines. Merely because the current target is a PDP11 (say) does not mean that code branches that are not executed should not contain 32-bit integer constants.

2. The LRM does not state if values in number declarations should be held with greater precision than MAX_DIGITS. As noted above, a small additional precision is necessary. Since there should be no effective limits on real literals (as lexical units), the case is rather different from integer literals. A very high quality compiler could produce a warning if excessive round off occurs in the calculation of a real expression. However, there is no language requirement for such a check (and indeed this is more likely to occur with run-time expressions which would need special tools). Note that the maximum required precision could be for fixed point if SYSTEM.MIN_DELTA is very small. This will happen if the maximum precision in floating point takes the same amount of space as for maximal precision fixed point (the extra space for the exponent in floating point will give fixed point an advantage).

3. The IEEE proposals for floating point processors have infinite values and also values which are 'Not a Number'. It is not clear how such values should be handled in Ada. For instance, 1.0/0.0 could be permitted in a literal expression with the value plus infinity. However, the arithmetic of infinite values varies in the IEEE proposal according to the 'mode' - the relevant ones here being projective (one infinity) and affine (signed infinity).

CHAPTER 5

Statements

5.1 Simple and Compound Statements

Semantic Ramifications

An empty sequence_of_statements must be coded as an explicit null_statement, since:

sequence_of_statements ::= statement {statement}

The only effect of the requirement that implicit declarations of labels, loop identifiers, and block identifiers occur in a certain order is to standardize error diagnostics for duplicate identifiers.

Compile-time Constraints

1. Within the sequence_of_statements of a subprogram, package, or task body (and excluding any nested subprograms, packages, or tasks), statement labels must be unique. (See 8.3.a.C/1.)

Test Objectives and Design Guidelines

1. Check that declarations cannot be interleaved with statements in a sequence_of_statements. Try them in a subprogram_body, package_body, block, if_statement, case_statement, and loop_statement.

2. Check that the statements in a sequence_of_statements are executed in succession. Avoid raising exceptions and avoid using exit, goto, and return statements. Check that labels on some of the statements in the sequence have no effect.

   Check that multiple labels are permitted on a statement.

   Check that labels are permitted at the beginning of a sequence_of_statements in every context that permits a sequence_of_statements, namely in accept statements, loops, case statement alternatives, exception handlers, if statement alternatives, package, task, and subprogram bodies, select statements, and select alternatives.

   Implementation Guideline: Use unique labels throughout these tests. (Checks for non-unique labels are performed in 8.3.a.T/1.)

3. Check that an empty sequence_of_statements must be coded as an explicit null_statement in all contexts that permit a sequence_of_statements. (See the preceding objective for a list of these contexts.)

5.2 Assignment Statements

Semantic Ramifications

Although the target variable and the source expression must have the same type (checked at compile-time), the expression may yield any value of that type, after which the value is checked to see if it satisfies the subtype constraints of the target variable. (If not, an exception is raised.) If the check fails, an exception is raised. This must occur before any portion of the target variable is updated. Hence, if assignment of a value would violate a variable's constraints, an exception must be raised before any portion of the variable is modified.

The assignment statement has pure "copy" semantics, i.e., the semantics are equivalent to:

1. Determine the identity of the target variable (i.e., evaluate and save the values of any target index expressions).

2. Evaluate the expression (and hold the entire value in a nonoverlapping temporary variable).

3. Check the constraints, i.e., check that the identity of the target variable is still valid, and check that the expression value satisfies the subtype constraints of the target variable.

4. Update the target variable with the evaluated expression value.

When assigning to scalar types (step 3 above), NUMERIC_ERROR, CONSTRAINT_ERROR, or any other exception must not be raised while checking to see if the expression's value is in the target variable's range. This applies in particular to INTEGER assignments when the range bounds or expression value are near INTEGER'FIRST or INTEGER'LAST.

Implementations may choose a different order for the above actions, provided that any exceptions to be raised must be raised before any portion of the target variable is updated.

Consider the following example:

```
type DISCRIM is (INT, BOOL);
type VR (D : DISCRIM) is
    record
        case D is
            when INT =>
                I : INTEGER;
            when BOOL =>
                B : BOOLEAN;
        end case;
    end record;

R : VR;
```

```
function F return INTEGER is
begin
        R := (D=>BOOL, B=>TRUE);
        return 1;
end F;

R := (D=>INT, I=>0);
R.I := F;        -- What happens here?
```

Step 1 could be done after step 2. Then R.I need be verified only once, namely after step 2. The CONSTRAINT_ERROR exception must be raised before step 4 for either order, and hence the example does not depend on the order of evaluation of R.I or F.

Now consider the following example:

```
type VR (D: DISCRIM) is
        .....   -- As above.

R : VR;

function F return INTEGER is
begin
        R := (D=>INT, I=>0);
        return 1;
end F;

R := (D=>BOOL, B=>TRUE);
R.I := F;        -- What happens here?
```

One order raises CONSTRAINT_ERROR in step 1. But if step 1 is done after step 2 then everything checks OK and the assignment is possible. Since the effect is different, depending on the order of evaluation, this example is an erroneous Ada program.

Compile-time Constraints

1. The types (but not necessarily the subtypes) of the target variable and the source expression must match.

2. The object being assigned to must not have been declared as a constant in an object declaration, as an in formal parameter, or as a component of such an object. It must not be a loop parameter, discriminant component of a record type, nor can it be a function_call, attribute, or operator symbol.

3. If the name being assigned to has the form of an identifier, the identifier cannot be the name of an enumeration literal, subprogram, entry, block, loop, statement label, package, pragma, task, or type

4. The type of the object being assigned to must be a type for which assignment is available, i.e., it cannot be a task type or limited private

type, nor can it be a composite type having a component of a type for which assignment is not available.

Exceptions

Exceptions raised during evaluation of the variable and expression are covered in Chapter 4.

1. CONSTRAINT_ERROR is raised if an assignment to a scalar variable would violate the variable's range constraint. (This check is suppressed by applying RANGE_CHECK to the type of the expression, to any object containing the variable being assigned to, or to the variable itself.)

2. CONSTRAINT_ERROR is raised for assignment of array types, see 5.2.1.E/1.

3. CONSTRAINT_ERROR is raised if an assignment to a constrained record or private variable (including a variable designated by an access value) tries to change the discriminant value of the variable. (The check for this exception situation is suppressed by applying INDEX_CHECK to the variable or its type; the check is suppressed for a given discriminant by applying INDEX_CHECK to that discriminant of the variable, or to the type of the discriminant.)

4. CONSTRAINT_ERROR is raised if the variable is of a constrained (array, record, private, or limited private) access type, the value being assigned is not null, and

   - any index bound of the designated object does not equal the corresponding bound specified for the access type's constraint. (This check is suppressed by applying INDEX_CHECK to the access type or, for a given index, to the type of the index.)

   - any discriminant of the designated object does not equal the corresponding value specified for the access type's constraint. (This check is suppressed by applying INDEX_CHECK to the access type or, for a given discriminant, to the type of the discriminant.)

Test Objectives and Design Guidelines

1. Check that an assignment_statement replaces the current value of the target variable with the value of the source expression. Check this for INTEGER, BOOLEAN, CHARACTER, FLOAT (separately), a fixed type (separately), another enumerated type, STRING, another array type, a non-variant record type, a variant record type, and an access type.

2. Check that the left side (target) of an assignment_statement must be a variable, i.e., cannot be an expression, such as a function call that returns an access value, or any of the forbidden kinds of named entities mentioned in C/2-4.

3. Check that multiple assignments are not permitted within a single

assignment_statement, in particular that the following forms are prohibited:

```
variable, variable := expression;
variable := variable := expression;
variable := expression operator (variable := expression);
```

4. Check that the types of the target variable and source expression must match at compile-time (see 5.2.C/1). Try combinations such as:

    INTEGER vs. FLOAT
    INTEGER vs. LONG_INTEGER
    INTEGER vs. CHARACTER
    array of INTEGER vs. array of FLOAT
    arrays and slices
    access record vs. record

5. Check that, for scalar types (INTEGER, BOOLEAN, CHARACTER, FLOAT (separately), a fixed type (separately), another enumerated type), the CONSTRAINT_ERROR exception is raised when the expression value is outside the target variable's range, and that the value of the target variable is not altered. Use variables and expressions that do not by themselves cause exceptions to be raised prior to the range check. Try some subtests that can be range constraint checked at compile-time and other subtests that must be checked at run-time. Use values that are just inside and just outside the target range (at both ends).

6. Check that the equality operator (=) cannot be used as an assignment symbol.

7. Check that NUMERIC_ERROR, CONSTRAINT_ERROR, or any other exceptions are not raised for INTEGER assignments where the expression value is actually in the target variable's range and where the target variable range bounds and/or the expression value are near INTEGER'FIRST or INTEGER'LAST. Use expressions whose evaluations, in themselves, will not raise exceptions.

8. Check that a record variable declared with a specified discriminant value cannot have its discriminant value altered by assignment. Assigning an entire record value with a different discriminant value should raise CONSTRAINT_ERROR and leave the target variable unaltered. Try both static and non-static discriminant values.

9. Check that a record variable desigated by an access value cannot have its discriminant altered, even by a complete record assignment, and even though the target access variable is not constrained to a specific discriminant value. In other words, check that attempting to change the target's discriminant raises CONSTRAINT_ERROR and leaves the target record unaltered. Try both static and non-static discriminant values.

10. Check that record assignments use "copy" semantics. In particular, try the following:

```
R : record
        X, Y : INTEGER;
    end record;
R := (0,0);
R := (X=>1, Y=>R.X);
...
R := (X=>R.Y, Y=>2);
```

11. Check index and discriminant constraints for assignment of access subtypes, as in:

```
type T is access an_unconstrained_type;
subtype S1 is T constraint1;
subtype S2 is T constraint2;
W : T;
X1, X2 : S1;
Y1, Y2 : S2;
```

Check that

- any of the above variables can be assigned to each other if the value being assigned is null.
- X1 or X2 can be assigned to each other or to W.
- CONSTRAINT_ERROR is raised if X1 is assigned to Y1 and X1 is not null.
- CONSTRAINT_ERROR is raised if W is assigned to X1, W is not null, and the constraints of the object designated by W do not equal the constraints imposed on S1.
- null can be assigned to any of these variables.

Gaps

1. The rule in 5.2.0 stating when CONSTRAINT_ERROR is raised is different from the rule given in 5.2.1 for array assignments. We have assumed that the rule in 5.2.1 takes precedence over the rule in 5.2.0 for arrays, but it would be clearer if 5.2.0 said explicitly that a modified rule applied to array assignments.

5.2.1 Array Assignments

Semantic Ramifications

An array type includes the fact that it is an array, the number of dimensions, the type of each dimension (including constraints on permissible bounds values), the component type, and the component subtype constraints (LRM 3.6). An index constraint is used to form an array subtype whose bounds are constrained to the specified values (LRM 3.6). Such a subtype cannot be futher constrained.

The type of a slice is the base type of the original array, but the subtype (i.e., the index constraint) is different if the bounds are different (LRM 4.1.2).

When checking that the lengths of corresponding dimensions match, and the lengths do match, but happen to exceed INTEGER'LAST or SYSTEM.MAX_INT, then the length checking must succeed and must not raise NUMERIC_ERROR or CONSTRAINT_ERROR.

The semantics of assigning overlapping slices is equivalent to first assigning the expression value to a non-overlapping temporary variable and then assigning the temporary to the target.

Note that since only index constraints can be applied to an array type, only the index constraints may differ for different objects of a given array type. Thus, since the component type constraints must be the same for all objects of the array type, it is not necessary to check the component constraints in an array assignment. Any component constraint checking of the source expression value will already have been performed as part of the expression evaluation semantics, which must yield a value of the target array type. In particular, if the source expression is an array aggregate (see LRM 4.3.2), then the individual component values are constraint checked as the (temporary) array value is formed, after which the array value is assigned to the array target variable.

Note that an implementation may directly assign the components of an array aggregate to the components of an array target (without constructing a temporary array value), provided that all of the component values are constraint checked (and exceptions raised) before any of the target variable is updated (see 5.2.S).

Note that given these declarations,

```
type DAY is (MON, TUE, ..., SUN);
NORM : array (MON..FRI) of INTEGER;
```

then NORM := (WED..SUN => 0) does not raise CONSTRAINT_ERROR.

Exceptions

1. CONSTRAINT_ERROR is raised for the assignment of a non-null array value to a non-null array variable if the number of components for corresponding dimensions is not equal. (This check is suppressed by applying LENGTH_CHECK to the array variable, to the array variable's type, or (for a given dimension for an array) to the type of the index for that dimension.

2. CONSTRAINT_ERROR is raised if a non-null array value is to be assigned to an array variable having a null index range for one of its dimensions, or if a null array value is to be assigned to a non-null array variable. (This check is suppressed by applying LENGTH_CHECK to the array variable or its type or to the type of an index for which a null range is specified.)

Test Objectives and Design Guidelines

1. Check that the lengths must match in array and slice assignments. Try all meaningful combinations of the following:

   a. Match vs. mismatch.

      1. Lengths Match -- Check that all components are correctly assigned to the proper positions. For lengths that exceed INTEGER'LAST, check that the length checking does not raise an exception, such as NUMERIC_ERROR or CONSTRAINT_ERROR.

      2. Lengths Mismatch -- Check that an exception (CONSTRAINT_ERROR) is raised and that the target variable is not altered.

   b. Static vs. non-static.

      1. Static -- Use static bounds and indices so that the length check can be done at compile-time.

      2. Non-static -- Use non-static bounds and indices whose values can only be known at run-time so as to force a run-time length check.

   c. Arrays and slices.

      1. Array -- Use an entire array, i.e., don't subscript or slice it. Try both non-null and null arrays. For the source expression, try both named arrays and array aggregate constructors.

      2. Slice -- Try both non-null and null slices.

      Also try to mix these within single assignment statements (as to kind of target and/or source) whenever the types match. For example:

```
S1 : STRING(1..10);
S2 : STRING(1..20);
.....
S1 := S2(6..15);
```

2. Check that the assignment of overlapping source and target variables (including arrays and slices in various combinations) satisfies the semantics of "copy" assignment, i.e., is equivalent to first copying to a non-overlapping temporary variable. Try both static and dynamic bounds so as to allow and prevent compile-time detection of the overlap. Also try the following "overlapping" aggregate constructor assignment:

```
A : array (1..4) of INTEGER := (1..4 => 0);
A := (1, A(1), A(1), A(1));
```

   Also try the following "overlapping" concatenation of slices:

```
S : STRING (1..10) := "ABCDEFGHIJ";
```

```
S := 'K' & S(1..2) & S(1..2) & S(1..5);
```

Also try the above for an array of integers. Each kind of overlap should be tested in both directions, i.e., target bounds less than source bounds and target bounds greater than source bounds.

5.3 If Statements

Compile-time Constraints

1. The expressions appearing in conditions must be of the predefined type BOOLEAN. They must not be of types derived from BOOLEAN.

Test Objectives and Design Guidelines

Short circuit evaluation of conditions is tested in 4.5.1.T.

1. Check that every if_statement must end with END IF, even when statements are nested unambiguously.

2. Check that ELSE cannot precede ELSIF in an if_statement.

3. Check that ELSIF cannot be spelled as ELSEIF, ELSF, ELIF, or ELSE_IF.

4. Check that the expressions appearing in conditions must be of type BOOLEAN. In particular, try the integer literal zero (0) and a type derived from BOOLEAN.

5. Check that control flows correctly in basic non-nested if_statements that have no ELSIF or ELSE parts.

   Implementation Guideline: Since this form of if_statement is heavily used in the other executable tests to conditionally invoke the FAILED routine, it must be very carefully tested here. Thus, the order of the following subtests is critical.

   First do all of the following subtests where the condition has the value false. Each subtest has the form:

```
IF false_condition THEN
     FAILED ("message");
END IF;
I := I + 1;
```

   This checks that the THEN action is correctly skipped. The variable I should be initialized to zero before the first subtest. Its purpose here is to verify that control skips to whatever follows the THEN action.

   Use the following increasingly complex forms for the conditions: a boolean literal (e.g., FALSE), a named boolean constant or variable (e.g., B), a simple relation (e.g., A > 0), a logical expression (e.g., B AND A >

0), and a condition with AND THEN or OR ELSE. Do not use conditions that depend on short circuit evaluation.

All of the above subtests must be done twice: first with static conditions, and second with non-static conditions.

Next, all of the subtests (for all the above forms of conditions and for both static and non-static conditions) must be repeated, but with the conditions having the value true. Each of these subtests has the form:

```
        IF true_condition THEN
              I := I + 1;
              GOTO Ln;
        END IF;
        FAILED ("message");
<<Ln>>  NULL;
```

This checks that the THEN action is correctly initiated.

Next, do the following subtest:

```
IF static_true_condition THEN
      I := I + 1;
END IF;

IF dynamic_true_condition THEN
      I := I + 1;
END IF;

IF I /= n THEN
      FAILED ("message");
END IF;
```

where n is an integer literal whose value is the number of false_condition subtests plus the number of true_condition subtests and I is initially 0. This, combined with the previous false_condition and true_condition subtests, checks that the THEN action is either skipped correctly or is initiated and completed correctly (for non-null actions).

Lastly, try two subtests where the sequence_of_statements is the null_statement, one with a static_true_condition and one with a dynamic_false_condition.

6. Check that control flows to the correct alternative sequence_of_statements in complex non-nested if_statements. Try at least the following forms:

   a. IF ...
      ELSE ...
      END IF;

   b. IF ...
      ELSIF ...

```
        ELSIF ...
        END IF;

    c.  IF ...
        ELSIF ...
        ELSIF ...
        ELSE  ...
        END IF;
```

Implementation Guideline: Each of the above forms must be tried repeatedly for each possible control flow choice. Thus (b) above needs four subtests, one for each of the four choices: (1) IF selected, (2) first ELSIF selected, (3) second ELSIF selected, (4) none selected. Occasionally, an alternative sequence_of_statements, whether selected or not, should consist of a null_statement.

For the conditional expressions (conditions), start out with simple boolean variables (e.g., B) or constants (e.g., TRUE). Then try simple relational expressions (relations, e.g., A > 0). Then, try logical expressions (e.g., A > 0 AND B). Lastly, try conditions involving AND THEN and OR ELSE, but don't use conditions that depend on short circuit evaluation. However, it is not necessary to try each of these condition forms with each of the if_statement forms.

All of the above subtests must be done twice: once with static conditions and once with dynamic conditions.

It can be assumed that the basic non-nested if_statement with no ELSIF or ELSE parts and with the condition forms described above works correctly (see 5.3.T/5). Thus, this basic form can be used to verify whether the above forms work, and to conditionally invoke the FAILED routine when they have failed.

7. Check that control flows correctly in simple nested if_statements.

Implementation Guideline: Try about four subtests involving doubly or triply nested if_statements. Use mixtures of the forms of if_statements and of static and dynamic conditions mentioned in 5.3.T/5 and 5.3.T/6. Also try some null alternatives, i.e., where the sequence_of_statements is a single null_statement, sometimes selected and sometimes not. It can be assumed that non-nested if_statements work correctly (see 5.3.T/5 and 5.3.T/6).

8. Check that control flows correctly in complex nested if_statements.

Implementation Guideline: This is intended to be a worst case test in terms of probable actual user programs. Try a nesting of if_statements that is at least ten deep along the selected path and at least five deep along a few of the nonselected paths. Along the selected path, use a mixture of all of the forms of if_statements and of static and dynamic conditions mentioned in 5.3.T/5 and 5.3.T/6. Likewise, use mixtures of these forms in the nonselected paths. The alternative sequences_of_statements should include the following forms:

a. a null_statement only,

b. an assignment_statement only,

c. an if_statement only,

d. at least one assignment_statement and at least one if_statement in various orders.

It can be assumed that non-nested if_statements work correctly (see 5.3.T/5 and 5.3.T/6).

9. Check that neither an if_statement nor any of its alternative sequences_of_statements implicitly introduces a new name scope. In particular, check that an alternative cannot begin with a declarative_part, that names declared just outside an if_statement cannot be declared as labels inside the if_statement, and that labels declared in one alternative of an if_statement cannot be declared as labels in another alternative of the if_statement. (See also 8.3.a.)

5.4 Case Statements

Basic properties of case statements are treated in 5.4.a. Interactions between the subtype of a case expression and the set of values covered by the alternatives are checked in 5.4.b.

Note that the vertical bar in the syntax production is intended as a terminal symbol, not a meta-syntactic symbol.

5.4.a Basic Case Statement Properties

Semantic Ramifications

The scope of statement labels inside case statements is discussed in Chapter 8. Constraints on the use of goto statements are discussed in 5.9 and Chapter 8.

Case expressions must have a discrete type, i.e., an integer, predefined-enumeration type, user-defined enumeration type, or a user-defined type derived from a discrete type. In particular, private types implemented with discrete types cannot be used in case expressions outside the scope of the declaration implementing the type. Also strings (even strings of length one) and fixed point types with integral DELTAs are not permitted.

Note that all forms of choice are permitted in case statements, in particular, choices of the form:

```
ST range L .. R
ST
A'RANGE(n)
```

Note that choices are limited to static simple_expressions; hence relational and logical operators cannot be used in choices unless the expression containing these operators is parenthesized. Of course, relational operators are only permitted if the case expression is a BOOLEAN expression, since overloaded relational operators cannot be used in static expressions (see 4.9).

If a programmer wishes to obtain the effect of executing several alternatives for a single case expression value, he will have to write a statement of the form:

```
case I is
   when 1 => goto    FIRST;
   when 2 => goto    SECOND;
   when 3 => goto    THIRD;
   when others => goto REST;
end case;
   <<FIRST>>     ... ; goto REST;
   <<SECOND>>    ... ; -- fall through to THIRD
   <<THIRD>>     ... ; goto DONE;
   <<REST>>      ... ;
   <<DONE>>      ...
```

An implementation may wish to provide special optimizations for such a program structure.

An implementation may wish to use a jump-table implementation for alternatives whose non-*others* choices cover a small range of values and an if-then-else implementation when a large range of values is covered but only a few alternatives are present.

Note there is no constraint prohibiting vacuous choices, e.g., a choice denoting a null range of values, a choice whose value is outside the subtype of the case expression (see 5.4.b), or an *others* choice that can never be selected.

It is only required that choice values have the *type* of the case expression. For derived types, this means values are permitted as (vacuous) choices that cannot actually be assigned to variables having that type. For example, given the usual definition of the type DAY, consider the derived type:

```
type WEEKDAY is new DAY range MON .. FRI;
```

According to LRM 3.4, this is equivalent to:

```
type !WEEKDAY is new DAY;
subtype WEEKDAY is !WEEKDAY range MON .. FRI;
```

where *!WEEKDAY* is an anonymous type name. Hence, SAT and SUN are values of type *!WEEKDAY*, although these values cannot be assigned to a variable of type WEEKDAY, since they do not satisfy WEEKDAY's range constraints. Since vacuous

choices are permitted, the following case statement, although bizarre, is legal:

```
X: WEEKDAY;
...
case X is
   when SAT | SUN =>  ...        -- vacuous alternative
   when MON..WED =>   ...
   when THU..FRI =>   ...
end case;
```

Compile-time Constraints

1. The type of the case expression and each choice must be the same.

2. The case expression must have a discrete type.

3. Every choice must contain only static expressions, i.e., for choices of the form V, L .. R, and ST range L .. R, L, R, and V must be static expressions, and for choices of the form ST, where ST is a subtype name, ST'FIRST and ST'LAST must be static expressions.

4. Two choices must not have a value in common.

5. An others choice, if present, must be the only choice given in the last alternative specified for a case statement.

6. The type of the case expression must be determinable independently of the values or types of the choices used in the case statement.

Exceptions

Note that CONSTRAINT_ERROR can be raised by static discrete ranges (see 3.6.1, LRM 10.6, and Gaps).

Test Objectives and Design Guidelines

1. Check that:

- the reserved word is is required;

- when cannot be replaced by if,
  | cannot be replaced by or,
  => cannot be replaced by then,
  end case cannot be replaced by end, endcase, or esac,
  is cannot be replaced by of;

- when the case expression is a simple variable, the name of the variable cannot follow end case;

- when a case statement is labeled, the label name cannot follow end case;

- the others choice must be the only choice given in the last alternative.

  Implementation Guideline: Try an others choice as the first and middle alternative, and try it as the first, middle, and last choice in a set of choices for the last alternative.

2. Check that an alternative must have at least one statement (see 5.1.T/3) and that it can consist of one or more statements (implicitly checked by other tests in this Section).

3. Check that the following types are permitted as the type of a case expression (see also T/4 and T/5):

   - BOOLEAN
   - CHARACTER
   - user-defined enumeration type
   - INTEGER
   - user-defined types derived from these types

   Implementation Guideline: Use an integer type, an enumeration type, and a derived discrete type in forming the derived types for this test.

4. Check that a private type implemented with a discrete type can be used in a case expression within the scope of the declaration implementing the type.

5. Check that non-discrete types are unacceptable. In particular, check strings of length 1, fixed point types with integral DELTAs, and private types implemented as discrete types but used outside the scope of the declaration defining the private type's implementation.

6. Check that static expressions (other than simple literal or constant name values) are allowed as case expressions (see also T/21).

7. Check that a variable used as a case expression is not considered local to the case statement. In particular, check that the variable can be assigned a new value, and the assignment takes effect immediately (i.e., the case statement does not use a copy of the case expression).

   Implementation Guideline: Use a discriminant of a variant record as the case expression and change the discriminant by assigning a new record to the variable. Then check that the discriminant field has the correct value (e.g., by attempting to access components unique to the new variant part).

Checks involving discrete ranges

20. Check that

- the type of the case expression and each choice must be the same;
- every pair of choices must cover a disjoint set of values.

Implementation Guideline: Use both single values and ranges of values, and check for overlapping values within a single alternative and between alternatives. Use overlapping ranges whose end points are different, e.g., 3..5 and 4..6 as well as ranges in which overlap occurs only at the end points. Use some examples in which a large range of values has to be checked for potential overlap. The choices should not all occur in monotonically increasing or decreasing order.

21. Check that non-static choice values are forbidden.

Implementation Guideline: Try a variable whose range is restricted to a single value. Try a discrete_range of the form ST, where ST is a subtype name having at least one non-static bound, as well as choices of the form ST range L .. R and L .. R, where either L or R is non-static.

22. Check that all forms of choice are permitted in case statements, and in particular, that forms like ST range L .. R, ST, and A'RANGE(n), where A is an array with static bounds, are permitted.

Implementation Guideline: Use the same subtype name in more than one choice. For the form ST range L .. R, try at least one subtype name with non-static bounds (see Gaps).

23. Check that choices using constant names (including subscripted constant names) are permitted.

24. Check that choices denoting a null range of values are permitted, and that for choices of the form ST range L .. R where L > R, neither L nor R need be in the range of ST values.

Check also that an others alternative can be provided even if all values of the case expression have been covered by preceding alternatives (see also 5.4.b).

Implementation Guideline: The vacuous alternatives should have null as its sequence of statements in one test and a non-null sequence of statements in a separate test.

25. Check that out of range derived type values are permitted as vacuous choices, e.g., using the WEEKDAY example discussed earlier.

26. Check that choices within and between alternatives can appear in non-monotonic order.

27. Check that relational, membership, and logical operators are allowed as choices only if the expressions containing these operators are enclosed in parentheses.

28. Check that CONSTRAINT_ERROR is raised for non-null choices of the form ST <u>range</u> L .. R if ST has static or non-static bounds and either L or R is outside ST's bounds (see 3.6.1.T/4).

<u>Optimization</u> <u>checks</u>

41. Check that the flow of control in a case statement is appropriate.

<u>Implementation</u> <u>Guideline</u>: Use a case statement with a small enumeration type inside a loop and check that each aernative is executed in the appropriate sequence. Also check a case statement whose alternatives contain only goto statements.

42. Check that a case statement may have:

- a large number of potential choices grouped into a small number of alternatives, e.g.,

  INTEGER'FIRST .. -101
  -100
   100
  100..INTEGER'LAST
  <u>others</u>

- a small range of choices grouped as a small number of alternatives (suggesting a jump-table implementation), e.g., alternatives using values in the range 1 .. 10

- a sparse set of alternatives in a large range, e.g., 1, 2, 1_000, 5_000, 10_000 .. 10_200.

- a few alternatives covering a large range, e.g., 1 .. 10_000, -9_000..0, 11_000..INTEGER'LAST.

- a small range very far from 0, e.g., 10_001, 10_002, 10_003, 10_004; (this permits a biased jump-table implementation)

- an <u>others</u> alternative that covers a small range of non-contiguous values, e.g., MON and FRI;

- an <u>others</u> alternative that covers several non-contiguous ranges of values, e.g., 1 .. 10, 15, 1000, -500 .. -400, etc.

and that the appropriate alternatives are executed.

Overloading Checks

60. Check that if the case expression is an overloaded literal or function call, the case statement is considered illegal.

Implementation Guideline: Try a function overloaded with an INTEGER and FLOAT return value, a function returning a value of an INTEGER and a derived integer type, and an integer literal in a context containing a derived integer type.

Gaps

1. 5.4 says "The values specified by choices given in a case statement must be determinable statically." A similar statement in 3.7.3 refers to 4.9 for a definition of when choices are determinable statically. However, 4.9 does not really state if the discrete range ST range L .. R is a static choice if both L and R are static but ST is not a static subtype. Note that in this case, the validity of the discrete range cannot be determined until run time, i.e., if the value of L or R is not within the range of ST's values, CONSTRAINT_ERROR presumably must be raised. We have assumed that since L and R are static, the choices are determinable statically, but it could be argued that the intent was to require no run-time checking of choices at all, in which case, our interpretation would be incorrect.

5.4.b When others Can Be Omitted

An others alternative can be omitted if every value in the subtype of the case expression is covered by non-others choices. In essence, if the expression is a primary, the subtype of the expression is the subtype of the primary; otherwise, the subtype is the type of the expression, independent of the subtypes of primaries in the expression.

Since we are only concerned with scalar expressions as far as case statements are concerned, the forms of primary of interest here are:

- literal
- variable or constant
- function_call
- qualified_expression, and
- (expression), where expression is one of these primaries

Hence, if we have:

```
I: INTEGER range 1 .. 5;
subtype FIVE is INTEGER range 1 .. 5;
J: FIVE;
```

```
function F return FIVE;
```

then any of the following case statements need only have choices covering the range 1 through 5, although choice values outside this range are permitted:

```
case I is
case J is
case F is
case (F) is
case FIVE'(I+1) is
```

Note in particular that the expression I + 0 has the subtype INTEGER, and hence, <u>case</u> I+0 <u>is</u> must provide choices for all values in the range of INTEGERs. However, FIVE'(I+0) is constrained to have a value in the range 1 through 5, and so only choices covering these values are required.

When the subtype range is non-static, the range of the base type is used. Hence:

```
subtype HUNDRED is INTEGER range 1..100;
subtype FIFTIES is HUNDRED range 1..M;
```

is non-static, if M is not a static expression. A case expression with subtype FIFTIES must cover the range of the base type of FIFTIES, i.e., INTEGER (not HUNDRED; see 3.3).

There is an important interaction between the use of a loop parameter as a case expression and the need for an <u>others</u> alternative, e.g., consider:

```
for I in FIFTIES range 1 .. 10 loop
   case I is
```

the subtype of I is static (see Gaps) and hence, only the values 1 .. 10 need be covered.

The subtype of an expression is never contextually modified, e.g., in the inner case statement of the following example:

```
for I in 1 .. 100 loop
   case I is
      when 1 .. 10 =>
         case I is
            -- must still cover 1 .. 100
```

and hence, the inner case statement must still cover the range of I's static subtype (i.e., 1 .. 100) before <u>others</u> can be omitted.

Note that if the subtype of the case expression has a null static range, <u>no</u> case alternatives are needed at all, e.g.,

```
for I in 1..0 loop
   case I is
```

end case; -- legal!

When an implementation discovers that an others alternative has incorrectly been omitted, it would be helpful to users if the values of the missing alternatives were indicated in a diagnostic message. However, this is not a language requirement.

Compile-time Constraints

1. The others alternative must be present if the set of values covered by the set of the choices does not cover 1) the set of values associated with the subtype of the case expression, when the subtype is defined with static bounds; otherwise, 2) the set of values associated with the base type of the expression.

Test Objectives and Design Guidelines

1. Check that if a case expression consists of a literal, variable, function invocation, qualified expression, or a parenthesized expression having one of these forms, and the subtype of the expression is static, an others can be omitted if all values in the subtype's range are covered, and must not be omitted if one or more of these values are missing.

   Implementation Guideline: Use subtype names with both static and non-static bounds, when possible. The interaction between loops and case statements is tested separately below and in 5.5.T/11-14.

2. Check that if a case expression consists of a variable, function invocation, qualified expression, or a parenthesized expression having one of these forms and the subtype of the expression is non-static, others can be omitted if all values in the base type's range are covered, and must not be omitted if one or more of these values are missing.

   Implementation Guideline: Do not use loop parameters as case expressions in this test (these are tested in 5.5).

3. Check that when the case expression is a loop parameter, an others alternative can be omitted under the appropriate circumstances (see 5.5.b.T/12,13).

4. Check that even when the context indicates that a case expression covers a smaller range of values than permitted by its subtype, an others alternative is required if the subtype value range is not fully covered.

   Implementation Guideline: Use the nested case statement example at least.

5. Check that if the case expression is I+0, the full range of INTEGER values must be covered if I is an INTEGER type or an integer subtype.

Gaps

1. The concept of a static subtype is not really defined in Section 4.9. In particular, it is not clear that if ST is a subtype with non-static bounds, the declaration

   I: ST range L .. R;

   implies that I has a static subtype if L and R are static. We have made this assumption, however.

5.5 Loop Statements

The discussion of loops is divided into four subsections:

a. properties of all loops

b. FOR loops

c. WHILE loops

d. continuous loops

5.5.a Properties of All Loops

Compile-time Constraints

1. The identifier following end loop is present if and only if the loop is labeled, and if present, the identifier must be the same as the loop label.

Test Objectives and Design Guidelines

1. Check the basic syntactic requirements:

   - a loop can be labeled, and if labeled, the same label must be present at the end of the loop; if not labeled, no label is permitted at the end of the loop;

   - the label at the end of the loop cannot be a name of the form P.LABEL, where LABEL is the name of the loop;

   - the label at the end of the loop cannot be a loop parameter name or a statement label;

     Implementation Guideline: For the statement label case try

     <<A>> loop ... end loop A;

   - when a labeled loop is nested inside a labeled loop, the inner

loop cannot be terminated by an end loop that mentions just the outer loop label;

- the forms

```
loop for I in 1..10; ... end loop;
loop while X; ... end loop;
```

are prohibited;

- the reserved word loop cannot be replaced by do or a semicolon as in

```
for I in 1 .. 10 do .. end [for];
while A do ... end [while];
for I in 1 .. 10; ... end;
while A; ... end;
```

- a loop cannot be terminated just with end (i.e., end loop is indeed required);

- check that forms

```
for I in 1..10 while A loop ... end loop;
while A for I in 1..10 loop ... end loop;
```

are not permitted;

- check that a loop body cannot be empty (see 5.1.T/3);

- check that a loop body can consist of more than one statement (note: this check is performed implicitly as a result of coding other tests required for loops);

- check that several levels of loop nesting are permitted. Design a capacity test to ensure an arbitrary degree of loop nesting is permitted.

- check that a loop parameter cannot be a subscripted variable, a record component, or a selected name identifying a variable in the scope enclosing the loop.

5.5.b FOR Loops

A loop introduces a new scope for loop_parameters, but not for labels appearing inside the loop. See 8.3.b and 8.3.a for further discussion.

If a loop parameter is renamed inside a block in a loop, the new name shares the properties of the loop parameter, namely, the new name cannot be used in an assignment context. See 8.5 for further discussion.

The type and subtype of the loop_parameter are determined by the discrete_range. Section 3.6.1 gives the rules for determining the type and subtype of a discrete_range. The subtype of a loop parameter is only of interest when the loop parameter is used in a case statement, because the subtype of the case expression determines when an <u>others</u> alternative can be legally omitted (see 5.4).

Consider the following situation:

```
for I in L .. R
     case I is
```

What is the largest and smallest range of values that must be covered by the case statement alternatives?

The answer to this question is determined by expanding the loop statement into the form:

```
for I in T range L .. R
     case I is
```

and then treating I in the case statement as though it had been declared:

```
I: T range L .. R;
```

T is determined by the <u>type</u> of L and R, without regard to the subtype of L and R (see 3.6.1). Moreover, if L and R are both composed solely of integer literals, then T is the predefined type INTEGER.

Given that the type of I is determined, then the legality of the case statement follows the rules discussed in 5.4. In particular, the legality of case statements having no <u>others</u> alternative is affected by whether L and R are both static expressions (see Gaps, 5.4.b).

An implementation must be careful to avoid raising NUMERIC_ERROR or CONSTRAINT_ERROR when evaluating loops of the form:

```
for I in INTEGER'LAST-10 .. INTEGER'LAST loop ...
for I in reverse INTEGER'FIRST .. INTEGER'FIRST + 10 loop ...
```

When evaluating null ranges, care must also be taken if the absolute difference between the bounds can exceed INTEGER'LAST, e.g.,

```
for I in INTEGER'LAST .. INTEGER'FIRST loop ...
```

Note that an enumeration type can be given a representation such that successive components of the type do not have successive integer values, e.g.,

```
type E is (A, B, C, D);
for E use (3, 10, 50, 1000);
...
for J in A .. C loop
```

J must take on the values A, B, C successively. In effect, an implementation could implement such a loop as

for J' in E'POS(A)..E'POS(C) loop

and then wherever J was mentioned in the original loop, the implementation would supply E'VAL(J').

Compile-time Constraints

See also the constraints for 3.6.1.

1. The loop parameter must not appear in an assignment context, i.e., as the target of an assignment statement or as an in out or out parameter.

Exceptions

Note that evaluation of the discrete_range can cause an exception to be raised (see 3.6.1.a; also see Gaps).

Test Objectives and Design Guidelines

The tests for discrete_range (Section 3.6.1) are repeated here to ensure that discrete_ranges used in loops are processed correctly.

1. Check that a loop_parameter cannot be used as the target of an assignment statement or as an actual in out or out parameter.

2. Check that if a loop_parameter is renamed, the new name cannot be used as the target of an assignment statement or as an actual in out or out parameter. (See 8.5.T/?)

3. Check that the loop_parameter is assigned values in ascending order if reverse is absent, and descending order if reverse is present. (Note: loops over enumeration types with user-defined representations are tested in T/14.)

4. Check that the loop is not entered if the lower bound of the discrete range is greater than the upper bound, whether or not reverse is present.

   Check that the loop bounds are evaluated only once, upon entry into the loop.

   Implementation Guideline: Use both static and dynamic bounds. At least one test should specify bounds with fairly complex arithmetic expressions. Attempt to modify the loop bounds by changing the value of a variable or subscript used in the discrete_range.

5. Check that loops whose upper bound is INTEGER'LAST (for non-reverse loops) and whose lower bound is INTEGER'FIRST (for reverse loops) are executed without raising NUMERIC_ERROR or CONSTRAINT_ERROR.

6. Check that loops can be specified for BOOLEAN, CHARACTER, INTEGER, user-defined enumeration types, and types derived from these types. Use all three forms of discrete_range. Include types derived from derived types.

7. If an implementation supports LONG_INTEGER or SHORT_INTEGER, check that loops using literals and variables of these types can be written.

   Implementation Guideline: Note that in these cases the type of the literal must be indicated explicitly, by writing one of the following forms:

   ```
   for I in SHORT_INTEGER range 1 .. 10 loop
   for I in 1 .. SHORT_INTEGER'(10) loop
   ```

8. Check that integer literals outside the range of INTEGER cannot be used in a loop of the form for I in L .. R loop.

9. Check that the type of a loop_parameter is correctly determined. In particular, for a loop of the form

   ```
   for I in L .. R loop
   ```

   where L and R are integer literals, check that I has the type INTEGER and cannot be used in a context requiring a value of a type derived from INTEGER or requiring a LONG_INTEGER or SHORT_INTEGER value.

   Implementation Guideline: Separate tests should be written for LONG_INTEGER and SHORT_INTEGER.

10. Check that if L or R are overloaded enumeration literals, the overloading is properly resolved and the loop_parameter is considered to be of the appropriate type (see Overloading Resolution in 3.6.1).

11. Check that the type of a loop_parameter is correctly determined for loops of the form:

    ```
    for I in ST range L .. R loop
    ```

    In particular, if ST is a subtype of T, check that I can be assigned to variables declared with some other subtype of T as well as to variables of type T.

    Check that the above form is accepted even if L and R are both overloaded enumeration literals such that L .. R would be ambiguous if ST were omitted.

12. Check that the subtype of a loop_parameter is correctly determined so that when the loop_parameter is used in a case statement, an others alternative is not required if the choices cover the appropriate range of subtype values. Use loops of the form

for I in ST range L .. R loop

where

- L and R are both static expressions, and ST is a static subtype covering a range greater than L .. R. (The case statement alternatives need only cover the range L .. R);

- L and R are both static expressions but ST was defined with at least one non-static bound (see Gaps, 5.4.b). (The case statement alternatives need only cover the range L .. R);

- L or R is a non-static expression, but ST is a static subtype. (The case statement must cover the range ST'BASE'FIRST .. ST'BASE'LAST.)

  Implementation Guideline: Check that it is illegal for case alternatives to cover just the range L .. ST'BASE'LAST when L is static, or ST'BASE'FIRST .. R when R is static).

13. Using a case statement, check that in loops of the form:

    for I in L .. R loop

    the subtype of I is equivalent to a loop of the form:

    for I in T range L .. R loop

    where T is INTEGER if L and R are integer literal expressions.

    Implementation Guideline: Use both integer expressions and enumeration literals for L and R.

14. Using a case statement, check that in loops of the form:

    for I in ST loop
    for I in A'RANGE loop

    the subtype of I is ST'FIRST .. ST'LAST or A'FIRST .. A'LAST, respectively.

    Implementation Guideline: Use A'RANGE for multidimensional arrays as well as single dimension arrays.

15. Check that if a discrete range of the form ST range L .. R raises an exception because L or R is a non-static expression whose value is outside ST's range of values (or because ST'FIRST is non-static and L is static and less than ST'FIRST; similarly for ST'LAST and R), control does not enter the loop before the exception is raised.

16. Check for correct processing of iterations over an enumeration type whose representation is user-defined as a non-contiguous set of integers. Use both reverse and normal loops.

Implementation Guideline: Use 'IMAGE or UNCHECKED_CONVERSION to check that the proper representations are assigned to the loop parameter.

Gaps

1. The specification for a null discrete range in 3.6.1 says the upper bound of the discrete range must be the predecessor of the lower bound or else CONSTRAINT_ERROR is raised. This restriction was probably intended only for discrete ranges used in index constraints, entry declarations, and slices. It was probably not intended to apply to discrete ranges in iteration clauses, since <u>for</u> I <u>in</u> 1 .. N <u>loop</u> would raise CONSTRAINT_ERROR if N was not equal to zero.

5.5.c WHILE Loops

Compile-time Constraints

1. The expression following <u>while</u> must have the predefined type BOOLEAN.

Test Objectives and Design Guidelines

1. Check that the expression following <u>while</u> cannot have a type derived from BOOLEAN.

2. Check that if the <u>while</u> expression is statically or non-statically FALSE when the iteration_specification is evaluated, the loop is not entered.

   Check that the <u>while</u> condition is evaluated before each iteration, i.e., that if an assignment changes the value of the <u>while</u> expression, the loop is exited the next time the condition is evaluated.

   Implementation Guideline: Change the value of the condition by assigning to a variable used in the expression and by changing the value of a subscript used in the expression.

3. Check that the <u>while</u> expression may be arbitrarily complicated for both static and non-static expressions.

5.5.d Continuous Loops

Test Objectives and Design Guidelines

1. Check that <u>loop</u> .. <u>end loop</u> is executed until control is explicitly transfered out of the loop (via an <u>exit</u>, <u>goto</u>, or <u>return</u> statement) or implicitly by an exception.

5.6 Blocks

The declarative part of a block is tested in Section 3.9. The exception-handling part of a block is tested in Chapter 11. The fact that each of these parts is optional in the presence of the other is also tested there. The fact that both these parts may be missing at the same time, leaving the block in the simple form

```
begin
     <sequence_of_statements>
end
```

will be tested here.

Compile-time Constraints

1. If a block identifier is given, the identifier in the end statement closing the block must be present and must be the same as the label of the block.

Test Objectives and Design Guidelines

Transfers of control within a block and out of a block are tested in Section 5.9. The detection of (illegal) transfers of control into blocks is tested in Chapter 8.

Redeclaration of identifiers and name identification issues are treated in Section 8.3.

1. Check that:

   - a named block must be closed by an end statement which repeats the block identifier.

   - more than one identifier cannot be given as the name of a block.

   - the optional parts of blocks, if present, may still be empty, as in:

     ```
     declare
     begin
             <sequence_of_statements>
     exception
     end;
     ```

     but the non-optional part, the <sequence_of_statements>, must not be empty. (It may consist of nothing but the "empty" statement null, though.) Note: the requirement for a null statement is checked in 5.1.T/3.

   - ```
     declare
     exception
     ```
     ```

```
        when ... => ...;
    end;
```

is forbidden.

- blocks can be embedded in blocks.

  Implementation Guideline: As a capacity test, check at least 65 levels of nesting. There is no need to have declarative parts in these blocks.

- the syntax used for blocks in other block-structured languages

```
    begin
         <declarative_part>
         <sequence_of_statements>
    end
```

  is not valid in Ada. (See 5.1.T/1.)

2. Check that blocks can have declarative parts and that the declarations in these parts have an effect limited to the block in which they appear.

5.7 Exit Statements

Compile-time Constraints

1. The innermost construct enclosing an exit statement must be loop-end loop, and not the begin-end of a subprogram, package, or task body, or the do-end of an accept statement.

2. The loop_name in an exit statement must be the name of an enclosing loop statement.

Test Objectives and Design Guidelines

1. Check that:

   - exit statements cannot be written outside a loop body.

   - that an exit statement cannot transfer control out of a subprogram, package, task, or accept statement.

   Implementation Guideline: These tests should be performed inside a block wholly contained in the body of a loop.

   Check that the loop parameter cannot be used as a loop_name in an exit statement.

   Check that an exit statement with a loop_name must be enclosed by a loop statement with the same name.

2. Check that a simple exit unconditionally transfers control out of the innermost enclosing loop. Three contexts should be considered -- the statement is:

    1. one of the simple statements in the <sequence_of_statements> constituting the body of the loop;

    2. inside a block in the body of the loop; exits should be attempted from the body of the block and from an exception handler for the block (see 11.2.T/7 for the exception handler test);

    3. inside a compound statement in the body of the loop.

3. Check that the exit statement condition is evaluated each time through a loop, and that it is evaluated correctly whether positioned at the beginning, middle, or end of the loop.

4. Check that an exit statement with a loop_name terminates execution of the enclosing loop statement with the same name, as well as all inner enclosing loop statements.

5.8 Return Statements

Semantic Ramifications

A subprogram's exception handler may contain a return statement that returns from the subprogram (see LRM 11.2).

The restriction on where return statements can appear boils down to forbidding return statements in package or task bodies. However, the restriction must be carefully checked to insure the following examples are illegal:

```
procedure P is
    package Q is ... end;
    package body Q is
    begin
        return; --illegal
    end;
begin
    declare
        package QQ is ... end;
        package body QQ is
        begin
          return;      -- still illegal
        end;
    begin
      ...
    end;
end P;
```

The above examples, of course, are illegal even if the return statement is nested in the statement part of a block inside the package bodies. So the relevant consideration is to ensure that the innermost construct enclosing a return statement is the begin-end portion of a subprogram body, not a package or task body.

Compile-time Constraints

1. The innermost construct enclosing a return_statement must be the begin-end portion of a subprogram or the do-end portion of an accept statement, not the begin-end portion of a package body or task body.

2. Return_statements in functions must have expressions. Return_statements in procedures must not have expressions.

3. The type of a return_statement's expression must match the return type specified in a function's specification or declaration. (Note: the subtypes do not have to match.)

Exceptions

1. If the subtype constraint of a function's return value is not satisfied by the value in a return statement, CONSTRAINT_ERROR will be raised under the same conditions this exception would be raised when assigning the return value to a variable having the function's subtype. The exception is raised at the place of the return statement, within the function body.

Test Objectives and Design Guidelines

1. Check that a return_statement cannot appear outside the body of a subprogram or accept_statement.

   Implementation Guideline: After trying simple cases, try to return from a package body and a task body where the package or task is nested in a block contained in a subprogram. Also try putting the package in the declarative part of a subprogram.

   Check that a return statement is permitted in an exception handler (see 11.2.T/5).

2. Check that the return_statements in functions must have expressions. Check that the return_statements in procedures cannot have expressions.

   Implementation Guideline: Use procedures nested in functions and vice versa.

3. Check that the type of a return_statement's expression must match the return type specified in a function's specification or declaration.

   Implementation Guideline: The subtype of the return expression should be different from the subtype of the function.

4. Check that a return_statement terminates execution of the innermost enclosing subprogram or accept_statement. Check that for functions, the value specified in the return_statement is actually returned. Check these for both recursive and non-recursive subprograms. (Return statements in exception handlers are checked in 11.2.T/5.)

5. Check that the constraints on the return value of a function are satisfied (or CONSTRAINT_ERROR is raised) when a function returns control to its invoker. Use a subtype name in the function specification or declaration that is more restrictive than the subtype name for a variable used as the return statement expression. Try some subtests where the return statement's expression value satisfies the function return value subtype constraints (which must return control normally), and some subtests where the constraints are not satisfied (which must raise an appropriate exception, see 5.8.E/1).

6. Check that if a return statement raises an exception, it can be handled within the body of the function without necessarily propagating the exception to the function's invoker.

5.9 Goto Statements

Semantic Ramifications

Although a goto statement cannot transfer control from outside into a subprogram, package, task, exception handler, or compound statement (i.e., an if, loop, case, accept, or select statement, or a block) this restriction does not necessarily imply these constructs form name scopes for labels (see 8.3.a).

Compile-time Constraints

1. A goto statement cannot attempt to transfer control out of a subprogram, package, task, or accept statement.

2. A goto statement cannot attempt to transfer control into a compound statement, subprogram, package, task, or exception handler, or between the alternatives of a case statement or if statement, or between statements belonging to different exception handlers in a sequence of handlers.

3. The label referenced in a goto statement must be defined in the innermost enclosing subprogram, package, or task containing the goto statement (see 8.3.a.C).

4. A goto statement in an exception handler cannot transfer control into the sequence of statements preceding the set of handlers.

Test Objectives and Design Guidelines

1. Check that

- a goto statement cannot reference nonexistent labels (see 8.3.a.T/4).

- a statement with multiple labels can be a target of a goto statement, and any one of its labels may be used for this purpose.

- a goto statement cannot transfer control out of a subprogram, package, task, or accept statement (see 8.3.a.T/4).

- a goto statement cannot transfer control into a compound statement or from the body of a block into a handler for that block.

- Check that jumps between branches of a case statement or if statement are not permitted.

2. Check that jumps out of an exception handler (but not to a label defined in the statements guarded by the exception handling part of a block) are allowed.

   Check that jumps from an exception handler in a unit to another belonging to the same unit are not allowed.

   Check that jumps out of compound statements (other than an accept statement) are possible.

CHAPTER 6

Subprograms

6.1 Subprogram Declarations

Semantic Ramifications

The restrictions on the use of variables, functions, and allocators in the declaration of parameters does not apply to the result type of a function. It is not clear whether this is an oversight or not (see Gaps). The intent of the restriction is to ensure that even if the formal parameter declarations are elaborated more than once in a list of declarations, the same values will be produced for constraints and for initialization expressions.

Note that a default value can be provided for an unconstrained array formal parameter type.

Note that the object designated by a constant access object is not a constant. Hence, such objects cannot be named in parameter declarations.

Note that although a parameter declaration may not use the name of a parameter declared earlier in the same formal part, it may use the name of a parameter declared later in that part if such a name has a meaning outside the parameter declaration, e.g.,

```
C: constant INTEGER := 5;
function F(A, B : INTEGER := C; C : INTEGER) return INTEGER;
```

Note that an attribute of an in parameter can be used in an initial value and that the name of the function can be used in specifying such an attribute name:

```
function F(X: INTEGER := F.X'LAST) return ...;
```

Compile-time Constraints

1. An operator_symbol used as a designator in a function declaration must not be one of the following strings: "/=", "in", "not in", "and then", "or else", or ":=" (see 6.7).

2. The subprogram specification must not hide a subprogram (or other identifier) previously declared in the same declarative part (see 8.3.C/?)

3. A function subprogram must only have parameters of mode in (see 6.5).

4. Formal parameter identifiers of subprograms must be distinct from each other and from identifiers declared in the subprogram's declarative part or a preceding generic part (see 8.3.e.C/1).

5. Initialization of formal parameters is permitted only for parameters with

mode in and only for subprograms that are not designated with operator symbols (see 6.7). An initialized formal parameter must not be of a type for which assignment is not available, i.e., a limited private type, a task type, or a composite type having a component of a type for which assignment is not available.

6. The initialization expressions and subtype indications in parameter declarations can only contain static expressions, names of constants, and attributes.

7. The subtype indication in a parameter declaration and subprogram specification must not use the name of any formal parameter declared previously in the same formal_part, nor may the name of the current parameter be used.

8. The operator symbols "and", "or", "xor", "=", "<", "<=", ">", ">=", "&", "*", "/", "mod", "rem", and "**" must only be used in function specifications having two parameters (see 6.7).

9. The operator_symbols "+" and "-" must only be used in function specifications having one or two parameters (see 6.7).

10. The operator_symbol "not" must only be used in a function specification having a single parameter (see 6.7).

11. Formal parameters for the "=" operator must only be 1) of the same limited private type, or 2) of the same array or record type having a component whose type is (directly or indirectly) limited private (see 6.7).

12. An in out or out parameter cannot be specified for a task type (see 9.2).

Exceptions

1. CONSTRAINT_ERROR is raised if the value of the initialization expression does not satisfy the constraints of the subtype_indication (see 3.2.C/?).

Test Objectives and Design Guidelines

1. Check that certain syntactic malformations are forbidden, viz:

   - the identifier returns cannot be used in place of return in a subprogram_specification.
   - the reserved word return cannot be used in the specification for a procedure;
   - an operator_symbol for a one-character operator cannot be written as a character literal, e.g., '+';
   - parameter declarations cannot be separated with commas;
   - the mode designation out in is forbidden as well as in in and out out.

- a formal part for a function or procedure cannot have the form "()";

- a task type cannot be declared as an <u>in</u> <u>out</u> or <u>out</u> formal parameter;

- a subprogram cannot be declared as an object, a type, or a formal parameter;

- an array type definition is forbidden in a formal parameter declaration.

2. Check that "in", "not in", "and then", "or else", "/=", and ":=" are not permitted as operator_symbols in subprogram declarations (see 6.7.T/1).

   Check that all the permitted operator_symbols can be used in function specifications with the required number of parameters (see 6.7.T/2).

   Check that functions for "and", "or", "xor", "=", "<=", "<", ">=", ">", "&", "*", "/", "mod", "rem", and "**" cannot be declared with one or three parameters (see 6.7.T/1).

   Check that functions for "not" cannot be declared with two parameters (see 6.7.T/1).

   Check that functions for "+" and "-" cannot be declared with zero or three parameters (see 6.7.T/1).

3. Check that duplicate subprogram specifications are not allowed in the same declarative part (see 6.6.T/1 and 6.7.T/2).

   Check that a subprogram declaration and subprogram body can be given separately in the same declarative part, but that the subprogram declaration must precede the subprogram body.

4. Check that duplicate formal parameter names are forbidden in a single formal_part and that a formal parameter, generic parameter, and a local variable or enumeration literal cannot have the same name (see 8.3.e.T/1).

5. Check that initialization is forbidden for formal parameters of mode <u>in</u> <u>out</u> or <u>out</u> and for formal parameters of mode <u>in</u> if they are of a limited private type (see 7.4.2), a task type (see 9.2), or a composite type having at least one component (including components of components) of a limited private or task type.

6. Check that an initialization expression and subtype indication in a parameter declaration must not contain the name of a variable (including a dereferenced access constant), a user-defined operator or function invocation, or an allocator, e.g., check that A = A is not permitted as a default value for a BOOLEAN parameter if A is a variable. Check that a subtype indication in the return specification of a function is permitted (see Gaps) to use a variable name or to invoke a user-defined operator or function.

7. Check that the subtype_indication in a parameter_declaration or in the return type of a function specification is not permitted to use the name of any formal parameter declared in the corresponding formal_part, even if the formal parameter name is known outside the subprogram specification (see also 8.3.e.T/2).

   Implementation Guideline: Try examples with array parameters where the length of the returned array is a function of the length of the argument arrays, as for concatenation. Also try examples where an attempt is made to constrain two array parameters to have the same bounds.

8. Check that CONSTRAINT_ERROR is raised if the value of the initialization expression does not satisfy the constraints of the subtype_indication.

   Implementation Guideline: Try an array parameter constrained with non-static bounds and initialized with a static aggregate, a scalar parameter with non-static range constraints initialized with a static value, and a record parameter whose components have non-static constraints initialized with a static aggregate.

9. Check that a static expression, constant name, or attribute name can be used in the subtype indication or initialization expression of a formal parameter, and that if used as an initialization expression, the appropriate value is used as a default parameter value when the subprogram is called.

   Check that an unconstrained record type (with and without default constraint values) and an unconstrained array type are permitted as formal parameter subtype_indications (see 6.3.T/4,5).

10. Check that an in out or out formal parameter can be declared with a limited private type or a composite type for which assignment is not available.

Gaps

1. It may be an oversight that the restrictions on the use of variables and functions apply only to parameter declarations and not to the subtype indication specifying the return type of a function.

6.2 Formal Parameters

Semantic Ramifications

Note that the rules permit a programmer to write a matrix addition procedure that is called with three identical parameters, e.g., MAT_ADD(A,A,A), since once an assignment is made to a component of the out parameter, it is never again necessary to read the out parameter component. Hence, an implementation must not forbid such calls. However, such a call would be erroneous for a matrix multiplication routine that did not store the results of the multiplication in a temporary local array.

Note that if a formal in parameter is an access type, assignments can still be made to components of the parameter, but not to the parameter itself.

Compile-time Constraints

1. A formal in parameter must not be used as an actual in out parameter, an actual out parameter, or as the target of an assignment statement.

Exceptions

See 6.4.1.

Test Objectives and Design Guidelines

1. Check that a formal in parameter cannot be used as the target of an assignment statement or as an actual parameter whose mode is in out or out.

   Implementation Guideline: Use a simple scalar in parameter, an array and a record in parameter (attempt to assign to a component of the parameter or use a component as an out actual parameter), and an access in parameter.

   Check that labels and subprograms can neither be declared nor passed as subprogram parameters.

2. Check that components of access in parameters (including the .all component) can be used as the target of an assignment statement or as an actual parameter of any mode.

3. Check that scalar and access parameters are copied.

   Implementation Guideline: Check this by calling subprograms of the form F(A,A) where the second parameter is an in out or out parameter. Assignments to the second formal parameter should not change the value of the first formal parameter, nor should direct assignments to the actual parameter change the value of the corresponding formal parameter. Check this for both procedures and functions.

   Implementation Guideline: Check that if an exception is propagated from a subprogram, the values of the actual scalar and/or access parameters are the values at the time of the calls, even if assignments were made to the formal parameters before the exception was raised.

4. Check that aliasing is permitted for parameters of composite types, e.g., that a matrix addition procedure can be called with three identical arguments, e.g., MAT_ADD(A,A,A);.

   Check that for unconstrained array formal parameters, the bounds of the formal parameter are determined by the bounds of the actual parameter, even if a default value is specified for the formal parameter.

Implementation Guideline: Check for all parameter modes. In addition to "normal" cases, check that null-array bounds are passed correctly, e.g., that a null string can be passed to a formal STRING parameter, and that when a null aggregate is passed, so are the bounds of the aggregate.

5. Check that discriminant values for record, private, and limited private actual parameters are passed to unconstrained formal parameters, even if a default value is specified for the formal parameter (see 6.4.1.T/6).

6.3 Subprogram Bodies

Semantic Ramifications

Note that INLINE can be specified after a subprogram body has been declared.

Compile-time Constraints

1. The designator at the end of a subprogram body, if present, must be the same as the designator used in the subprogram specification.

2. A subprogram_body is only permitted as a library unit or in the declarative_part of a package_body, block, subprogram_body, or task_body. (It is not permitted as a declarative_item in a package specification or task_specification (see 7.3 and 9.1).)

3. (To be devoted to the rules for identity of subprogram specifications for the same subprogram.)

4. The INLINE pragma must appear in the same list of declarative items as the subprogram specification to which the pragma applies, except that it may appear in the private part of a package specification when it names a subprogram declared in the visible part.

Test Objectives and Design Guidelines

1. Check that the designator at the end of a subprogram body must be the same as the designator used in the subprogram specification.

   Implementation Guideline: Use both an operator_symbol and an identifier. Check that selected component notation cannot be used, even if the selected component properly identifies the subprogram.

2. Check that a subprogram body is forbidden as a declarative item in a package_specification or task_specification (see 7.1.T/).

3. (Check of non-identical subprogram specifications.)

4. Check that a procedure without a return statement returns correctly.

5. Check that a null statement, at least, is required in the body of a subprogram (see 5.1.T/3).

6.4 Subprogram Calls

Semantic Ramifications

The semantic details of actual-formal parameter association are discussed in 6.4.1. This section is limited to the remaining aspects of subprogram calls.

Compile-time Constraints

1. The form F() can be used if and only if F is a function with no parameters or whose parameters all have default values.

2. For a function or procedure call with positional parameters only:

   - the number of actual parameters must equal the number of formal parameters; or

   - the number of actual parameters must be less than the number of formal parameters, and, if N parameters are omitted, the last N formal parameters must have default values specified for them;

   - the base type of the <u>ith</u> actual and formal parameter must be the same.

3. For a function or procedure call with both named and positional parameters,

   - the total number of actual parameters must not exceed the number of formal parameters;

   - omitted actual parameters must correspond to formal parameters for which default values were specified;

   - positional parameters must appear first;

   - a named parameter must not be specified for a formal parameter if an actual positional parameter is also given for that formal parameter;

   - the base types of corresponding formal and actual parameters must be the same.

4. The formal parameter name in a named parameter association must be identical to that of a formal parameter in the corresponding subprogram.

5. No duplicates are permitted among the formal parameter names used in an actual_parameter_part.

6. Check that a subprogram can be called recursively and that non-local variables and constants are properly accessed from within recursive invocations.

Exceptions

Exceptions are discussed in 6.4.1.

Test Objectives and Design Guidelines

1. Check that the form F() cannot be used if

- F is a procedure, even one with no formal parameters or whose parameters all have default values;
- F is a function with at least one parameter not having a default value.

Check that the form P (without parentheses) cannot be used if:

- P is a parameterless function; or
- P is a function whose parameters all have default values.

2. For functions and procedures having no default parameter values, check that the number of actual positional and named parameters must equal the number of formal parameters.

Implementation Guideline: Use calls that are valid except for the number of parameters. Check calls to procedures and functions. Use purely positional notation, purely named notation, and a combination of positional and named notation.

Check that parameterless subprograms can be called with the appropriate notation.

3. Check that for a mixture of named and positional notation, named parameters cannot be interleaved with nor precede positional parameters.

Implementation Guideline: Use a call in which the types of all the formal parameters are identical. Check that in a mixture of named and positional notation, a named parameter and a later positional parameter cannot be specified for the same formal parameter.

Check that two or more named parameters cannot specify the same formal parameter.

Check that the name used in a named parameter must only be a name of a formal parameter.

Check that a formal parameter in named parameter association is not confused with an actual parameter identifier having the same spelling (see 8.3.e.T/3).

Check that a named parameter cannot be provided for a formal parameter if a positional parameter has already been given for that formal parameter.

4. For functions and procedures having at least one default parameter, check that:

   - calls of the form F(A,,B) are forbidden, where the second formal parameter has a default value;

   - for a call using only positional notation, no parameters can be omitted unless the default parameters are at the end of the parameter list.

   - for a call using named notation, check that omitted parameters must have default values;

   - for a call using named notation, check that regardless of the order of the actual parameters, the correct correspondence with the intended formal parameter is achieved.

5. Check that the base types of the formal and actual parameter cannot be different.

6. Check that calls of the form F (A|B => 0) are not permitted even if A and B are both integer formal parameters of F.

6.4.1 Actual Parameter Associations

Semantic Ramifications

Note that the term "variable_name" includes an indexed_component, a selected_component, a slice, and a function_call; it excludes an allocator, operator_symbol, or name of a constant. Note that for use in the name of a variable, a function_call must return only an access value (see 5.2) and this value must be sliced, subscripted, or selected. Furthermore, note that although operators can be overloaded to deliver access values, if, say, "+", is overloaded to yield an access value, "+"(A,B).all can be used as a variable name (since it has the form of a function call).

When type conversions are used with actual in out and out parameters, the value of the actual parameter is converted to the base type of the formal parameter (see 4.6), and then the base type of the actual parameter is used when converting the value of the formal to the type of the actual parameter. The use of a subtype name in a type conversion does not affect whether CONSTRAINT_ERROR is raised. CONSTRAINT_ERROR can be raised for only two reasons:

- the converted actual value does not satisfy the constraints of the formal (does not apply to out parameters);

- the converted formal value does not satisfy the constraints of the actual parameter's variable.

For example, consider:

```
subtype ST is INTEGER range -5 .. 10;
A,B: FLOAT range -10.0 .. 130.0;
procedure F(X: in out INTEGER range 1..150;
            Y:    out INTEGER range 1..150);
...
F(ST(A), ST(B));
```

If the value of A is -5.0 before the call, CONSTRAINT_ERROR is raised because the converted value does not satisfy the range constraint of X. If the value of A is 120.0, CONSTRAINT_ERROR is not raised, since ST's range constraints are not used in type conversions (see 4.6; "the value of the expression is converted to the base type of the type mark") and 120 satisfies X's range constraint. Similarly, if the value of X on F's return is 20, the value of A becomes 20.0 and the range constraint of ST is not relevant; only A's range constraint is relevant.

If the value of B is -10.0 before the call, no CONSTRAINT_ERROR is raised, since B's value is not converted to INTEGER when F is called nor is it passed to Y. (Note, however, that the conversion to INTEGER must be expressed to ensure that the base type of the formal and actual parameters is the same.) If the value of Y is 20 when F returns, the value of B is set to 20.0 and once again, ST's range constraint is not relevant.

(Note: array type conversions work differently, since sliding occurs.)

Compile-time Constraints

1. An actual out or in out parameter must be either the name of a variable or have the form of a conversion applied to a variable.

2. The base types of the formal and actual parameters must be the same.

Exceptions

1. If the value of an in or in out actual scalar parameter does not satisfy the range constraint of the formal parameter at the time of call, or if the value of a formal out or in out scalar parameter does not satisfy the range constraint of the actual parameter at the time of normal subroutine return, then CONSTRAINT_ERROR is raised (at the place of the call).

2. If a formal parameter is a constrained record, private, or limited private type, CONSTRAINT_ERROR is raised if the discriminant values for the actual parameter do not equal those for the formal parameter.

3. If an actual parameter is an aggregate, CONSTRAINT_ERROR is raised if its value does not satisfy the relevant constraints of the formal parameter (see 4.3.2 for details).

4. If a formal parameter is a constrained array type, CONSTRAINT_ERROR is raised if the actual parameter does not have index bounds identical to the corresponding formal parameter's bounds.

5. If an in or in out formal parameter is a constrained access type, CONSTRAINT_ERROR is raised if the value of the actual parameter is not null and the constraints of the actual parameter do not equal the constraints of the formal parameter, i.e., for index constraints, the bounds values must be identical for each index, and for discriminant constraints, the discriminant values must be equal.

6. If an in out or out formal parameter is a constrained access type and the value of the formal parameter is not null, CONSTRAINT_ERROR is raised when the subprogram returns if the constraints of the actual parameter do not match the contraints of the formal parameter, i.e., for index constraints, corresponding bounds values are not identical and for discriminant constraints, corresponding discriminant values are not equal.

Test Objectives and Design Guidelines

1. Check that the expression corresponding to an out or in out parameter cannot be:

   - a constant, including an in formal parameter, an enumeration literal, a loop parameter, a record discriminant, and a number name (see 6.1.T/1, in part);
   - a parenthesized variable;
   - a function returning a record, array, private, scalar, or access type;
   - an attribute;
   - an aggregate, even one consisting only of variables;
   - a qualified expression containing only a variable name;
   - an allocator;
   - an expression containing an operator.

2. Check that the base types of corresponding formal and actual parameters must not be different (see 6.3.T/6.).

3. Check that type conversions for out and in out actual parameters are permitted and have the intended effect. In particular, check that:

   - real to integer and integer to real conversions are permitted;
   - if a subtype name is used, CONSTRAINT_ERROR is only raised for in out parameters if the actual value does not satisfy the formal parameter constraints and, for in out and out parameters, if the value of the formal parameter, after conversion, does not satisfy the range constraint of the actual variable; the range constraint of the subtype never causes CONSTRAINT_ERROR to be raised, since all conversions are in terms of the base type.

- array type conversions are permitted, and that if a constrained array type is used, ...

- record type conversions are permitted, in particular, among records having different physical representations, i.e., different T'SIZE attributes.

4. Check that CONSTRAINT_ERROR is raised under the appropriate circumstances, namely:

   - when the value of a scalar in or in out actual parameter does not satisfy the range constraint of the formal parameter;

   - when the value of a formal out or in out scalar parameter does not satisfy the range constraint of the actual parameter at the time of normal subroutine return;

   - when an actual record parameter has constraint values that are not equal to the contraint values of the formal parameter;

     Implementation Guideline: In particular, try an uninitialized constrained actual out parameter.

   - when an actual in parameter is an aggregate (see 4.3.2.T/?);

   - when an actual array parameter has different bounds for one dimension than is required by the formal parameter;

   - for a constrained array type ...;

   - when the index bounds of an actual array access object do not equal the index bounds specified for an access formal parameter;

   - when the discriminant values of an actual record access object do not equal the discriminant values specified for an access formal parameter;

   - when the discriminant values or index bounds associated with the value of an unconstrained formal access out or in out parameter do not equal the constraint values of a constrained actual access parameter, upon return from the subprogram.

   Implementation Guideline: If no parameter mode is mentioned above, check for all parameter modes.

   Check that CONSTRAINT_ERROR is raised at the place of the call (i.e., within the caller, not within the called subprogram) in the above circumstances.

5. Check that CONSTRAINT_ERROR is not raised under the appropriate circumstances. In particular, check that no exception is raised:

- at the time of call, when the value of an actual out scalar parameter does not satisfy the range constraints of the formal parameter;

- at the time of call, when an actual access parameter has the value null and the formal parameter is constrained (even if the subtype of the actual parameter does not match that of the formal parameter);

- on normal return, when the formal parameter value is null and the actual parameter is constrained (even if the subtypes of the formal and actual parameters are not the same);

- on normal return, when an actual out access object does not have constraint values equal to those of the formal access parameter.

  Implementation Guideline: Use "access to access" types as well as "access to non-access" types.

6. Check that unconstrained record, private, limited private, and array formal parameters inherit the constraints of the actual parameter.

   Implementation Guideline: For record, private, and limited private types having default discriminant constraints, be sure to try an uninitialized constrained variable as an out parameter.

   Check that assignments to (formal parameters of) unconstrained record, private, and limited private types without default constraints (i.e., for which T'CONSTRAINED is always true) raise CONSTRAINT_ERROR if an attempt is made to change the constraint of the actual parameter (by making a whole-record assignment to the formal parameter).

   Check that assignments to (formal parameters of) unconstrained record, private, and limited private types with default constraints (i.e., for which T'CONSTRAINED is true or false depending on its value for the actual parameter) raises CONSTRAINT_ERROR if the actual parameter is constrained and the constraint values of the object being assigned do not satisfy those of the actual parameter. Check that CONSTRAINT_ERROR is not raised if the actual parameter is unconstrained, even if the assignment changes the constraints of the actual parameter.

   Implementation Guideline: Try these checks for nested procedure calls as well, i.e., where an unconstrained formal parameter is used as an actual parameter in a subprogram call.

7. Check that actual parameters are evaluated and identified at the time of call, e.g., use a call of the form P(I, A(I)) where the parameters of P are out parameters.

   Check that for out parameters, the value of the actual parameter (except for discriminant values, if any) is not passed to the formal parameter.

8. Check that all permitted forms of actual parameters, including names that contain functions returning an access type, are permitted.

6.4.2 Default Actual Parameters

Semantic Ramifications

The material in this section of the Reference Manual is discussed in 6.4.0 of this Guide.

6.5 Function Subprograms

See also 6.1.

Compile-time Constraints

1. All parameters of a function subprogram must have the mode in.

2. A function body must contain a return statement specifying a return value, excluding any return statements in subprograms or packages declared within the function body.

Test Objectives and Design Guidelines

1. Check that in out and out parameters cannot be specified for functions.

   Check that a return statement with a value specified is required inside a function body (see 5.8.T/1,2,3).

   Implementation Guideline: Check that absence of any return statement at all, or presence of a single return statement without a return value, makes the subprogram definition illegal.

   Implementation Guideline: Check that a function definition is illegal if the only return statements it contains are inside subprogram definitions or packages nested in blocks contained in the function body.

2. Check that a function is accepted if its only return statements occur in unexecutable sections of code.

   Implementation Guideline: Use static expressions in if statements, case statements, and loops to produce unexecutable sections of code.

6.6 Overloading of Subprograms

Semantic Ramifications

Note that for packages, the visible and private part of a package specification and the declarative part in a package body are all considered to

form the same declarative part for purposes of deciding when conflicting declarations are present (see 7.1).

Compile-time Constraints

1. Two subprograms having the same name cannot be declared in the same declarative part if:

   - they are both procedures or both functions; and

   - the number, order, names, and base types of the parameters are the same; and

   - the same parameters have default values; and

   - for functions, the result base types are the same.

2. Ambiguous overloaded subprogram calls are forbidden (to be expanded).

Test Objectives and Design Guidelines

1. Check that subprogram redeclarations are forbidden. Use two subprograms in the same declarative part that are identical except for one of the following differences:

   - the parameters are named differently.

   - the subtypes of a parameter are different. In particular, check for different subtype names, different range constraints, different accuracy constraints, different index constraints, and different discriminant constraints;

   - the result subtypes of two functions are different. In particular, check for different subtype names, different range constraints, different accuracy constraints, different index constraints, and different discriminant constraints.

   - the parameter modes are different; also try reordering the parameters and changing their modes.

   - default parameter values are different;

   Implementation Guideline: Try these redeclarations, among others:

   - one declaration in the visible part of a package and the other in the private part or body;

   - one declaration in the private part and the other in the body;

   - both declarations in the declarative part of a package body.

   Check that a subprogram cannot have the same identifier as a variable,

type, subtype, constant, number, or array declared previously in the same declarative part.

2. Check that overloaded subprogram declarations are permitted in which there is a minimal difference between the declarations. In particular, use declarations that differ in only one of the following aspects:

   - one is function; the other is a procedure.

     Implementation Guideline: Try parameterless subprograms as well as subprograms having at least one parameter.

   - one parameter has a default value and the other does not.

   - one subprogram has one less parameter than the other (the omitted parameter may or may not have a default value).

   - the base type of one parameter is different.

   - one program is declared in an outer declarative part, the other is declared in an inner part, and

     - the parameters are ordered differently;

     - one subprogram has one less parameter than the other, and the omitted parameter has a default value;

   - the result type of two functions declarations are different.

   Implementation Guideline: Each of the subprograms in the above tests must be called, if possible, to insure the correct subprogram is invoked.

6.7 Overloading of Operators

Semantic Ramifications

Note that equality is not definable for task types, nor for composite types containing a component of a task type.

Compile-time Constraints

1. An operator_symbol used as a designator in a function specification must only be one of the following strings: "and", "or", "xor", "=", "<=", "<", ">=", ">", "+", "-", "&", "not", "*", "/", "mod", "rem", and "**".

2. The operator symbols "and", "or", "xor", "=", "<=", "<", ">=", ">", "&", "*", "/", "mod", "rem", and "**" must only be used in function specifications having exactly two parameters.

3. The operator_symbols "+" and "-" must only be used in function specifications having one or two parameters.

4. The operator_symbol "not" must only be used in a function specification having a single parameter.

5. Operator declarations must not have default values specified for their parameters.

6. The type of the formal parameters for the "=" operator must be the same limited private type or the same array or record type having a component (or a component of a component) whose type is a limited private type.

Test Objectives and Design Guidelines

1. Check that "in", "not in", "and then", "or else", "/=", and ":=" are not permitted as operator_symbols. In particular, check that "in" cannot be defined as a function with three parameters of the same type and that ":=" cannot be defined as a procedure with one out or one in (or in out) parameter.

   Check that functions for "and", "or", "xor", "=", "<=", "<", ">=", ">", "&", "*", "/", "mod", "rem", and "**" cannot be declared with one or three parameters.

   Check that functions for "not" cannot be declared with two parameters.

   Check that functions for "+" and "-" cannot be declared with zero or three parameters.

   Check that initialization expressions cannot be specified for operator symbol functions.

   Check that the parameter types for "=" cannot be different, nor can both parameters be of the same scalar, access, (non-limited) private type, or task type.

   Check that the type of the parameters for "=" cannot be an array type whose components are of a scalar, access, (non-limited) private, task type, or record type none of whose components (directly or indirectly) are of a limited private type.

   Check that the type of the parameters for "=" cannot be a record type none of whose components (directly or indirectly) are of a limited private type.

2. Check that all the permitted operator_symbols can be used in function specifications with the required number of parameters:

   - with two parameters -- "and", "or", "xor", "=", "<", "<=", ">", ">=", "&", "*", "/", "mod", "rem", "**", "+", "-".

   - with one parameter -- "+", "-", and "not".

   Check that these functions are invoked when the appropriate operator_symbol is used in an expression.

Check that except for "=", operator specifications can have parameters with different types.

3. Check that operators for the predefined types can be redefined, e.g., try redefining "+" with INTEGER arguments and returning an INTEGER result. Check that the redefined operator is invoked when infix notation is used.

CHAPTER 7

Packages

7.1 Package Structure

Semantic Ramifications

Although a package specification that contains no subprogram, task, or package specifications does not require a package body, one can be provided. The only useful work such a body can perform is to initialize variables declared in either its package specification or some other package specification. Such an optional package body poses no difficulties if a package is not separately compiled, but if the package specification is separately compiled, then some care needs to be taken. The details of interactions between packages and separate compilation will be covered in Chapter 10.

It is a consequence of the syntax (see 3.9 and 3.1) that neither a package body, task body, subprogram body, or body stub can appear in a package specification. Any such constructs must appear, if at all, in the corresponding package body (see 7.3).

Visibility rules for packages are discussed in 8.3.f.

Compile-time Constraints

1. If an identifier is present at the end of a package specification or package body, it must be the same as the package identifier.

Exceptions

The exceptions that can be raised when elaborating a package specification or a package body are those raised when elaborating any declarative part or those that are propagated (see 11.4.1(c)) from the package body.

Test Objectives and Design Guidelines

1. Check that if an identifier is present at the end of a package specification or body, it must be the same as the package identifier.

2. Check that package bodies, subprogram bodies, task bodies, and body stubs cannot appear in package specifications.

   Implementation Guideline: The package bodies should all appear at the end of the set of declarative items of either the visible or private part, and in the case of package and task bodies, a package and task specification should appear prior to the body declarations but not interleaved with them.

Check that a package body can be defined without a begin followed by a sequence of statements, or, if it contains at least one statement, that an exception reserved word is permitted even if there are no exception handlers. (Note: checks using exception handlers are provided elsewhere; see 11.4.T/1,3,7, and 11.)

7.2 Package Specifications and Declarations

Semantic Ramifications

Note that if a package specification is not separately compiled, the package can be named in at least two ways: directly, or as a selected component using the name of the enclosing unit, e.g.,

```
procedure P is
   package Q is
      -- both Q and P.Q refer to package Q
```

Note that subprogram specifications, package specifications, task specifications, object declarations, etc. can all appear in the private part. Of course, such entities are only known within the private part and the corresponding package body, except for private types declared in the visible part of the package.

LRM 3.9 states that a subprogram must not be called during the elaboration of a declarative part if its body appears later than the place of the call. The following example illustrates some of the situations that violate this restriction:

```
procedure P is
   package A is
      function AP return INTEGER;
      X: INTEGER := AP();         -- illegal
   end A;

   package B is
      function BP return INTEGER;
   end B;

   package body B is
      function BP return INTEGER is ... end BP;
      package NESTED is
         Z: INTEGER := A.AP();  -- illegal
         Y: INTEGER := BP();    -- legal now
      end NESTED;
   end B;

   package body A is
      Z: INTEGER := B.BP();     -- legal
      function AP return INTEGER is ... end AP;
   end A;
```

```
begin
   null;
end P;
```

Note that this rule applies to definitions of operators as well. Hence, if "+" is declared in A to take arguments of some type, it cannot be invoked until its body has been elaborated.

Note that an incomplete type declaration (of an access type, see 3.8) appearing in the visible part of a package must have a complete declaration given in the same visible part, or if it is given in a private part, its full declaration must appear in the same private part. For example, the following full declarations are illegal because they do not appear in the same list of declarative items as the incomplete type declarations:

```
package P is
   type T;
   type U;
   package Q is
      type U is INTEGER;     -- legal, but Q.U is different U from P.U
   end Q;
                         -- error since no complete declaration for T or U
private
   type T is INTEGER;        -- illegal
end P;
```

Similarly, a deferred constant declaration can only appear in the visible part of a package specification if it is of a private type declared previously in the same package specification (see 7.4).

Compile-time Constraints

1. If an incomplete access type declaration is given in the visible or private part of a package specification, the complete declaration must be given in the same visible or private part and neither in any nested package specification nor in the corresponding package body.

2. A subprogram whose specification is given in a package specification cannot be invoked until its complete definition has been elaborated in the corresponding package body.

3. A deferred constant declaration is permitted in the visible part of a package specification if and only if its type is a private type previously declared in the same visible part (see 7.4 and 3.2).

Test Objectives and Design Guidelines

1. Check that a package specification can be declared within a package specification.

Implementation Guideline: Don't use subprogram or task declarations

within either package specification (this situation is tested below in 7.3.T/1).

Check that a package body can be provided for a package specification that does not contain any subprogram or task declarations and that statements within the package bodies can be used to initialize variables visible within the package body.

2. Check that if an incomplete type declaration is given in the visible part, the full declaration cannot be given within the package's private part, nor can it be given in the visible part of a nested package specification.

3. Using packages declared in the declarative part of a subprogram body, check that a subprogram cannot be invoked before its body as been elaborated.

4. Check that the rules for deferred constant object declarations are satisfied (see 7.4.T/2).

5. Check that only declarative items can appear in the visible part of a package specification.

6. Check that representation specifications only can be given for the items declared in the package specification.

7.3 Package Bodies

Since according to 7.1, the body of any subprogram, package, or task declared in a package specification must appear in the corresponding package body, the following kinds of nestings can arise:

```
package A is
   C: constant INTEGER := 6;
   E: constant INTEGER := 4;
   procedure AP (X: INTEGER := C);

   package B is
      D: constant INTEGER := 5;
      procedure BP (X: INTEGER := C + D + E);
   end B;

end A;

package body A is
   D: INTEGER;

   package body B is
      E: INTEGER;
      procedure BP (X: INTEGER := C + D + A.E) is
         ...
      end BP;
```

```
      end B;

      procedure AP (X: INTEGER := C) is ... end AP;
   end A;
```

This example illustrates that:

1. the ordering of declarations in the body need not duplicate the ordering of the declarations in the specification;

2. the nesting of package specifications must be duplicated by the nesting of the corresponding package bodies (see 7.1);

3. the visibility rules for identifiers ensure that the default parameter initialization in BP's body declaration refers to the same entities that BP's specification referred to, since in attempting to determine what D denotes, one first looks in B's package body, then in B's package specification, then in A's package body, and then in A's specification (see 8.3). This sequence means that if A.E had been written as E, it would have been associated with a different entity than its association in the initial declaration of BP. Hence, a selected component form of name is needed to ensure that the two specifications of BP are "the same" (see 6.6).

Compile-time Constraints

1. If a subprogram or task specification has been given in a package specification, a package body must be provided in which the corresponding subprogram or task bodies are defined.

2. If a package specification, Q, is declared in a package specification, P, and package Q requires a package body, then a package body must be provided for package P as well.

3. A package body cannot be declared until its specification has been elaborated (see Gaps).

Test Objectives and Design Guidelines

1. Check that if a subprogram or task specification is provided in a package specification, a corresponding subprogram or task body must be provided in a package body.

   Check that if a package specification requires a body, and the package is declared inside another package specification, bodies must be provided for both specifications.

2. Check that the statements in a package body are executed after the declarations in the body have been elaborated.

3. Check that exceptions raised in the declarative or statement part of a package body are propagated properly (see 11.4.T/1,3,7, and 11).

4. Check that entities declared in a package body cannot be accessed from outside the package body (see 8.3.g.T/6).

5. Check that a null package body can be provided.

6. Check that a package body cannot be provided unless its specification has already been elaborated.

Gaps

1. The LRM does not clearly state that a package body cannot be provided without a preceding specification, although such a restriction applies to separately compiled packages (see 10.3) and is suggested by the wording of the first sentence in 7.1.

7.4 Private Type Definitions

Semantic Ramifications

In this section, we give ramifications, constraints, and objectives that apply to both non-limited and limited private types. Restrictions unique to these subclasses of private types are discussed in 7.4.1 and 7.4.2.

Note that although 7.4 seems to allow a deferred constant declaration for any private type, whether or not the type is declared in the same package as the constant, 3.2 states that deferred constants are permitted only for private types declared in the same private part as the deferred constant. Hence,

```
C: constant T;
```

is illegal unless a declaration for T appears in the same visible part containing the declaration of C. In particular, the following would be illegal:

```
package P is
   type T is private;
   package Q is
      C: constant T;              -- illegal
   private
      type T is ... ;             -- illegal
```

For both private and limited private types having discriminants, the only allowed variation in lexical structure between the private_type_definition and the full declaration is that names may be written differently if they denote the same entity. This means (see also Gaps, below) that selected component notation can be used to resolve naming difficulties, e.g.,

```
procedure P is
   X: INTEGER := 5;
   package R is
```

```
      type T(A: INTEGER := X) is private;
      X: INTEGER := 3;
   private
      type T(A: INTEGER := P.X) is ... ;
   end R;
```

Note that according to 8.3, it is necessary to write P.X in the full declaration of T, since X by itself would refer to R.X. Even though R.X and P.X have the same values at the time both declarations of T are elaborated, the second declaration would be illegal if we had not written P.X, since it would have referred to a different X than the one referred to in the first declaration.

The rule says that default values must be the same and the only variation allowed is in the form of names. Hence, if a default value were given as 3+2 in one declaration and as 2+3 in the other case, the second declaration would be illegal -- 3+2 and 2+3 are not names; they are literal expressions, and variation in literal expressions is not allowed. Note also that none of the expressions in the discriminant specifications for the full type declaration are evaluated -- it is only required that the names in each discriminant specification refer to the same declared entities.

Section 3.9 states that access to any entity before its elaboration is not allowed. This rule implies that deferred private constants cannot be used until their value has been defined in the private part, e.g., consider:

```
package P is
   type T is private;              -- 1
   C: constant T;                  -- 2
   procedure PP(X: T := C);        -- 3    -- illegal use of C
   V: T := C;                      -- 4    -- illegal use of C
private
   type U is ... ;                 -- 5
   type T is                       -- 6
      record
         A: U;                     -- 7
      end record;
   C: constant T := ... ;          -- 8
end P;
```

(3) and (4) are illegal since the constant has not yet been given a value. Note also that C cannot be declared in full until the full definition of T has been seen, and the amount of space allocated for C and V cannot be determined until T has been elaborated.

When determining the type of an object, it is always important to know whether the object is being referred to inside the package body that knows the definition of the private type. For example,

```
package P is
   type T is private;
private
```

```
   type T is new INTEGER;
end P;

package Q is
   type T_NAME is access P.T;
   X: T_NAME;
end Q;

package body P is
begin
   Q.X := new P.T(3);   -- legal (1)
end P;

package body Q
begin
   Q.X := new P.T(3);   -- illegal
end Q;
```

Reference (1) is legal since X.all is of type T, and within P, T is known to be represented as an integer. However, inside Q, T is private and so no values can be assigned to Q.X.all.

If a private type is defined by deriving it from another private type, the operations that are inherited by the derived type depend on the context of the definition. For example, consider:

```
package P is
   type T is private;
   type U is private;
   function "+" (X,Y: T) return  T;
   type V is new T;
private
   type T is new INTEGER;
   type U is new T;
end P;
```

The only predefined operations defined for objects of type U or V are assignment and equality. The "+" operator defined for T is not inherited by U or V (see 3.4) within the body of P.

However, if we define another package that uses package P:

```
with P;
package Q is
   type V is private;
private
   type V is new P.T;
end Q;
```

then within package body Q, objects of type Q.V inherit the P."+" operator for operands of type Q. In addition, within package body Q, the "+" for objects of type V can be named as Q."+" (see 3.4).

Note that all operators (except equality; see 6.7) can be overloaded or redefined to accept operands of a private type either within the package defining the private type or outside it. If the redefinition occurs within the package defining the private type, then the routine implementing "=" has access to the representation of the private type. If outside this package, however, the routine has the usual knowledge of a private type's representation, namely, no knowledge at all (except of discriminants, if any).

Equality can only be redefined for a limited private type (see 6.7) Composite objects having components of such a type do not make use of the new definition when equality comparisons are performed, since equality is not defined on a component by component basis (see 4.5.2). Note that even for equality, there is no requirement that the operator be specified in the package where its operand types are defined.

Compile-time Constraints

1. The innermost unit enclosing a private_type_definition must be the visible part of a package specification, not a subprogram body, task body, block, package body, or the private part of a package specification.

2. The full declaration of a private type declared in the visible part of a package specification must appear in the private part of the same package specification.

3. If a deferred constant declaration is given, it must appear in the visible part of a package specification containing a corresponding private type declaration and the full declaration of the constant must appear in the private part of this package specification.

4. If a private type definition is given for a type with discriminants, the corresponding full declaration must have exactly the same discriminant names and they must appear in the same order. Its subtype indications and default discriminant values (if any) must have the same lexical components in both declarations except that names can be written differently if they denote the same entity.

5. A full declaration of a private type cannot declare an unconstrained array type.

Test Objectives and Design Guidelines

1. Check that a private type definition (both limited and non-limited) cannot appear in the private part of a package specification, in a package body, subprogram body, block, or task body.

   Check that the full definition of a private type (limited and non-limited) cannot appear in the same visible part as the private type declaration, nor can it be omitted entirely from the package specification (even if it is provided in the corresponding package body).

   Check that a full definition of a private type cannot appear in the private part of a nested package specification.

2. Check that a deferred constant declaration cannot be given for a private type declared in some other package.

   Check that a deferred constant declaration cannot be given (for a private type) in the private part of a package (even the package containing the private type's declaration, whether or not it appears before or after the full declaration), nor can it be given in the corresponding package body, or a package specification nested within the package specification containing the private type declaration.

   Check that a deferred constant cannot be used as an initial value for an object declaration, constant declaration, or parameter declaration prior to its full declaration.

   Check that even if a private type has default values specified for all its components, a full declaration must be given for a deferred constant, and the full declaration cannot omit an initialization expression (see 3.2.C/3).

3. Check that for a private type declaration with discriminants, the full declaration must have the same discriminants specified, and that in particular, different literal expressions yielding the same value and lexically identical declarations that have different meanings are forbidden.

   Check that different selected component forms of name can be used in the corresponding declarations.

   Check that a full private type declaration cannot be an unconstrained array type.

4. Check that a full private (non-limited and limited) type declaration can be given in terms of any scalar, array, record (including record types with discriminants), or access type (including an access type with discriminants).

5. Check that within the package body, the representation of a private type is available, but that outside the package body, no predefined attributes or operators (except assignment, equality, and inequality, if the private type is non-limited) are available for objects of the private type unless explicit overloadings of these operators have been provided by the user.

6. Check that all forms of declaration (see 3.1) are permitted in the private part of a package specification except for deferred constant declarations, private_type_definitions, and package body, task body, and subprogram body declarations.

7. Check that subprograms can be declared outside the package that defines a (limited and non-limited) private type which use the private type as a parameter (of any mode) or as a result type.

8. Check that operator symbols can be overloaded (within a package that

defines a private type) to take arguments (or return results) of a private type and that such operators are invoked in place of any operators or other subprograms automatically inherited when the private type is defined as a derived type.

Check that overloaded operator definitions can be provided outside the package defining the private type.

Check that the equality operator can only be defined for operands of the same limited private type or same composite type having a component of a limited private type (see 6.7.T/2).

9. Check that if two private types are defined in the same package and one type is derived from the other, if operator redefinitions or subprograms are provided for the first type, the second type does not inherit this definition within the package body.

Gaps

It should be noted that renaming can introduce a different form of name that requires additional checking in discriminant specifications. This is probably an oversight, since the identity of names introduced by renaming cannot in general be determined at compile time, and no exceptions are associated with the processing required to determine that names are the same. For example, consider:

```
package R is
   C: array (1..5) of INTEGER := (1,2,3,4,5);
   type T(A: INTEGER := C(3)) is private;
   C_3: INTEGER renames  C(IDENT(3));
private
   type T(A: INTEGER := C_3) is ... ;   -- probably illegal
end R;
```

IDENT is a function that returns its argument value and is written so as to preclude compile-time analysis that this is its effect. C(3) is a name, and C_3 is a "different form of name", so the above declarations satisfy the wording of the specification, albeit, probably not its intent.

7.4.1 Private Types

Semantic Ramifications

In this section, we discuss those properties of private types that are unique to non-limited private types. The only difference between the two kinds of private type is that limited private types do not have the operations of assignment and equality automatically defined; non-limited private types do.

The first and second paragraphs in the LRM state that for "subprograms specified within the visible part, [assignment, predefined equality, and

predefined inequality] are the only externally available operations on objects of a private type." It should be understood that "operations" is used here to cover only those functions, procedures, and predefined operators that have access to the internal representation of a private type, as is explicitly noted in 7.4.2 for the more restricted class of limited private types -- "outside the defining package, subprograms having parameters of any mode can be defined for objects of a limited private type..."

The statement that the full declaration of a private type must be in terms of a type for which assignment and equality are available means that a non-limited private type cannot be defined in terms of a limited private type, or a type having a component for which assignment is unavailable. For example, if T is a private type being declared and U is a previously declared limited private type, then in

```
package P is
   type T is private;
   type U is limited private;
private
   type U is new INTEGER;
   type T is
      record
         X: U;  -- illegal
      end record;
end P;
```

the full declaration of T is illegal since it contains a component of a limited private type; assignment and equality would not be available for such a record type (see 7.4.2), and hence, the declaration of T is illegal. It doesn't matter that the representation of U is known when T is fully declared.

Compile-time Constraints

1. The full declaration of a non-limited private type must not use a type derived from a limited private ty; or a task type, nor can it be an array or record type with a component for which assignment is unavailable.

Test Objectives and Design Guidelines

1. Check that a private type cannot be (fully) declared as:

   - a type derived from a limited private type, task type, or a composite type for which assignment is unavailable;

   - a record or array type with a component of a limited private type, a task type, or a composite type for which assignment is unavailable.

   Implementation Guideline: Check that the above restrictions are enforced even if all the types are declared in the same package.

91 760 SOFTECH INC WALTHAM MASS F/G 9/2

ADA COMPILER VALIDATION IMPLEMENTERS' GUIDE.(U)

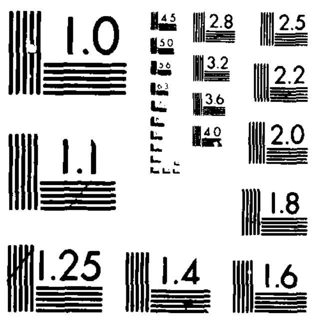
UN

OCT 80 J B GOODENOUGH MDA903-79-C-0687

CLASSIFIED 1067-2.3 NL

3 OF 4

AD A091760



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963 A

7.4.2 Limited Private Types

Semantic Ramifications

It should be noted that if a limited private type is used as an actual generic parameter, the formal parameter must be declared as a limited private type (see 2.3.2).

Note that although equality is not automatically provided for a limited private type, nothing forbids a programmer from overloading "=" for a limited private type. The assignment operator, ":=", cannot be overloaded, however.

Note that if "=" is overloaded for a limited private type, T, a composite type (e.g., an array of T's) still does not have equality defined, since equality for composite types is not defined in terms of equality of the component types (see 4.5.2). Equality can be defined for such composite types, however (see 6.7).

Note that assignment and pre-defined equality operations are permitted on limited private types within a package specification as soon as the full definition of the private type has been given. These operations are not available sooner because of the rule in 3.9 forbidding access to unelaborated entities. A variable or constant of a limited private type cannot have a value available for use until the full definition of the type has been elaborated. For example,

```
package P is
   type T is limited private;
   C: constant T;
   D: constant T;
private
   type T is new INTEGER;
   E: constant T := 5;          -- legal now
   C: constant T := 3;
   D: constant T := 4;
   CHECK: BOOLEAN := C = D;     -- legal now
end P;
```

Test Objectives and Design Guidelines

1. Check that objects of a limited private type cannot be compared for equality (or inequality) outside the package defining the type, nor can they be assigned to.

   Check that within the package defining a limited private type, assignment and equality comparisons are permitted, including within nested packages.

   Check that the restrictions on use of assignment and equality extend to objects having at least one component of a limited private type, and to types derived from limited private types.

2. Check that a variable declared outside the package defining a limited

private type cannot have an initialization specified explicitly (see 3.2.T/?).

3. Check that a parameter of a limited private type cannot have a default value specified if the subprogram specification appears outside the package defining the private type.

   Check that within the private part of a package (after the full declaration of a limited private type) or within the corresponding package body (including nested package bodies), a subprogram can be declared that has a parameter of a limited private type defined in the visible part of an enclosing package.

   Implementation Guideline: Among other tests, try a package specification nested in the private part of a package after the full declaration of a limited private type.

4. Check that no constant of a limited private type can be declared outside the package defining the private type, even if a value of the private type is accessible.

5. Check that an allocator for an access type designating objects of a limited private type cannot specify an initial value for the allocated object (see 4.8.T/2).

6. Check that a limited private type can be defined as a task type, as a type derived from a limited private type, or as a composite type that has a component for which assignment is not available.

7. Check that constants and variables of a limited private type can be declared and initialized in the private part of a package after the full declaration of the private type has been elaborated.

   Check that pre-defined equality is defined and available within the private part after the full declaration of the private type has been elaborated.

8. Check that user-defined equality is unavailable for a limited private type within the private part of the package declaring the type (since the body of the equality operator has not yet been elaborated).

CHAPTER 8

Visibility Rules

Semantic Ramifications

In this Section, we are concerned with the rules for determining the meaning of a particular identifier in an Ada program. The following aspects of the language are relevant to these rules:

- the nesting of constructs introducing a name scope. These constructs are:

  package specifications
  package bodies
  task specifications
  task bodies
  subprogram bodies
  blocks
  loops
  records

- the treatment of package specifications and corresponding bodies as part of the same name scope, even though these constructs are physically discontiguous

- access to names via WITH clauses

- access to names via USE clauses

- overloading of subprogram and enumeration identifiers

- identification of names in (separately compiled) subunits.

An important simplification in determining what meaning is associated with a given identifier is that the search for possible identifications is ordered:

1. lexically nested units are searched in an order reflecting how they are nested, except that task and package specifications are examined before the lexical context enclosing the corresponding body.

2. names in the STANDARD environment are considered;

3. then names accessed via WITH clauses and USE clauses are considered.

Potential overloading of identifiers does not affect the ordering of contexts considered.

The consequences of these observations are discussed in later sections.

8.1 Definitions of Terms

Semantic Ramifications

Ada distinguishes the scope of declarations and labels from the visibility of identifiers. At a given point in Ada text, an identifier may have potential associations with several declarations (i.e., several possible meanings). Which of these meanings is chosen is determined (for non-overloaded identifiers) by the visibility rules. This distinction between visibility and scope must be kept in mind in reading both this document and the Ada Standard. In essence, scope refers to the region of text over which a declaration has a potential effect, and visibility refers to the textual context in which a name has a particular meaning. (In some other languages, the term "scope" or "name scope" is used in the sense of "visibility" here.)

8.2 Scope of Declarations

The implications of this Section are discussed in conjunction with Section 8.3, since tests of the scope rules must make use of the visibility rules.

Gaps

1. The scope of attributes is not defined by the LRM.

8.3 Visibility of Identifiers

Interactions between scope and visibility rules are treated here in subsections addressing:

a. Labels

b. Loop Parameters

c. Records

d. Enumeration Literals

e. Parameters

f. Packages

g. Selected Component Names

h. Blocks

i. Access Types

Interactions with separate compilation are treated in Sections 8.4 and 8.5. Interactions with overloading are treated in Sections 6.6 and 6.7.

Gaps

1. Visibility rules for attributes are not specified in the LRM, although, as predefined entities, they probably don't need a rule. It would be reassuring if a Note were added to this effect.

8.3.a Labels

Semantic Ramifications

Section 5.1 states that statement labels must be unique within subprogram, package, or task bodies. This implies that if a statement inside a loop, block, or accept statement is labeled, a statement outside the construct (and in the same body) cannot have the same label. Hence, it is not adequate to enforce the restrictions on targets of goto statements (5.9) by treating loops, blocks, and accept statements as name scopes for labels, since this would imply labels inside and outside such constructs are distinct.

In addition, labels cannot conflict with other identifiers, e.g.,

```
procedure P (M : INTEGER) is
    L : FLOAT;
begin
    <<L>>       -- conflicts with FLOAT L
    for L in 1..10 loop        -- does not conflict with label L
```

Compile-time Constraints

1. Within the sequence_of_statements of a subprogram, package, or task body (and excluding any nested subprograms, packages, or tasks), statement labels must be unique.

2. For each label used in a goto_statement, there must be a corresponding statement label in the innermost enclosing subprogram, package, or task body.

3. Label identifiers must be lexically distinct (case is ignored; see 2.1) from other identifiers declared in the innermost enclosing subprogram, package, or task body.

Test Objectives and Design Guidelines

1. Check that a statement label inside a loop, accept statement, block body, or exception handler cannot be the same as a statement label outside these constructs.

2. Check that a label in a nested subprogram or module can be identical to a label outside such a construct. In particular, try a subprogram declaration in a block as well as a subprogram nested in a subprogram.

3. Check that a goto statement inside a loop, block body, or exception handler is permitted to access a labeled statement appearing before and after the loop or block, respectively.

4. Check that a goto statement in a nested subprogram, package, or task body is not permitted to reference a statement label in the outer subprogram, package, or task body.

5. Check that a loop parameter can be spelled the same as a label occurring prior to the loop or inside the loop. Similarly, check that labels cannot be spelled the same as formal parameters and variables.

8.3.b Loop Parameters

Semantic Ramifications

There are several consequences of the rule that the scope of a loop parameter extends only to the end of the corresponding loop:

1. The value of the loop parameter can be accessed only within the loop body.

2. Nested loops can have identically named loop parameters, in which case, the outer loop parameter is inaccessible within the inner loop unless the outer loop is labeled and selected component notation is used.

3. Non-nested loops can use the same identifier for their loop parameters but these identifiers are distinct loop parameters, i.e., may have different types.

4. A loop parameter can have the same identifier as some entity declared in the immediately enclosing scope.

In short, loops introduce a name scope for loop parameters. Note however, that loops do not introduce a name scope for labels (see 8.3.a, above).

Test Objectives and Design Guidelines

1. Check that the value of a loop parameter cannot be accessed from outside the loop body.

   Implementation Guideline: Attempt to access the loop parameter after the loop body and in a program containing no other use of the loop parameter identifier. (By accessing after the loop body, the implementation has a chance to insert the loop parameter identifier in its symbol table. If the insertion is not done properly, later references to the identifier will be successful.)

2. Check that:

a. nested loops can have identically named parameters with or without distinct types, and references in the innermost loop are associated with the innermost parameter, etc.

b. non-nested loops can have identically named loop parameters with distinct types, and that references within each loop are to its own loop parameter;

c. A loop parameter can have the same identifier as a variable declared in the scope immediately containing the loop.

Implementation Guideline: Use loop parameters containing more than a single letter, since JOVIAL gives single letter loop parameters the Ada interpretation, but gives a different interpretation to multiple letter loop parameters.

8.3.c Records

Semantic Ramifications

Components of a given record definition can have the same name as components of another record definition, i.e., two record types can both have components named X, either for nested or non-nested record type definitions.

The visibility rules for records imply that outside the record, the name of a component begins with the name of the record containing the component, and similarly, for components of components. Hence, if R is a record containing component S which contains component T, the only way to reference T outside the record definition itself is by writing R.S.T; R.T, T, and S.T are not valid references to this component.

Compile-time Constraints

1. A record component identifier within a given record definition must be uniquely named. In particular, components of different variants of a given record must have mutually distinct names.

Test Objectives and Design Guidelines

1. Check that two components of a given record definition must be uniquely named. In particular, check that components of different variants cannot have the same name, even if they have the same, statically defined, subtype. Since record discriminants are considered components of a record (see 3.7.1), check also that duplications are not permitted among the names of discriminants and the names of any other record components of a given record definition.

   Check that component names may be the same as names of other objects, viz., formal parameters, labels, loop parameters, variables, constants, subprograms, packages, tasks, and types.

2. Check that partial names for record components (as in PL/I) are not permitted, e.g., that if a record component is named T, and this is the only declaration for identifier T, the name T is not permitted outside the record as a reference to this component. Also, for a record of records, if the full name for a component is R.S.T, check that R.T is not permitted as a valid reference, assuming (of course) there is no T component of R.

8.3.d Enumeration Literals

Semantic Ramifications

Enumeration literals in outer scopes can be hidden by declarations in inner scopes. An inner declaration of a variable or some other non-overloadable entity hides an outer declaration of an enumeration literal using the same identifier, even if the type of the variable and the type of the enumeration literal are not identical.

Enumeration literals have the form of both identifiers and character literals (see LRM 3.5.1).

Compile-time Constraints

1. Duplicate enumeration literals (including character literals) are not permitted in a given enumeration type definition.

Test Objectives and Design Guidelines

1. Check that enumeration literals are hidden by inner declarations but not by additional enumeration declarations in the same scope, e.g., try a program like the following:

```
procedure Q is
   type FORT_INT is (I,J,K,L,M,N);
   type MY_INT_LIT is (K,L);
   X: FORT_INT;
   Y: INTEGER;
begin
L: declare
      M: INTEGER;        -- hides outer M
      N: FORT_INT;       -- hides outer N
      function I return INTEGER is ... end;     -- Q.I not hidden
      function J return FORT_INT is ... end;    -- Q.J not hidden
   begin
      N := K;             -- variable N, not literal N
      N := I;             -- literal I, not function I
      N := M;             -- illegal; Q.M not visible
      M := I();           -- variable M, function I
      Y := M;             -- variable Y, integer M
      X := M;             -- illegal; Q.M not visible
      X := N;             -- Q.X := L.N
      X := J;             -- Q.X := Q.J
```

```
            X := J();           -- Q.X := L.J();
            X := Q.J;           -- literal J
         end;
      end;
```

2. Check that duplicate enumeration literals (including character literals) are not permitted in a single enumeration type definition.

   Implementation Guideline: Use duplicates which differ only in the case of the letters as well as lexically identical literals.

8.3.e Subprogram Parameters

Semantic Ramifications

Subprogram parameters are considered to be entities declared immediately within the subprogram. LRM 8.3 states that such entities can be written as a selected component within such subprograms, as long as the name of the subprogram is not hidden. This implies that formal parameter identifiers must be distinct from each other and from identifiers declared in the subprogram's declaration list.

The use of a formal parameter in a named association does not hide an entity with the same name used as the actual parameter, e.g., calls of the form F(A => A) are permitted, where the first A is F's formal parameter name and the second A names an identifier visible in the context of the call to F.

Compile-time Constraints

1. Formal parameters of subprograms must be distinct from each other and from identifiers declared in the subprogram's declarative part.

2. A parameter value or name cannot be used in a constraint or default value appearing later in a parameter list, nor can it be used in the result type of a function declaration (see 6.1.C/7).

Test Objectives and Design Guidelines

1. Check that duplicate formal parameter names in a single parameter list are forbidden and that a formal parameter and local variable, generic parameter, or enumeration literal cannot have the same name.

2. Check that within a subprogram, a formal parameter can be used directly in a range constraint, a discriminant constraint, an index constraint, and an exception handler, but it cannot be used in a constraint or default value appearing in the parameter list, nor in the result specification for functions (see 6.1.T/7).

3. Check that a formal parameter in named parameter association is not confused with an actual parameter identifier having the same spelling.

4. Check that different subprograms can have parameters with the same name and/or type.

   Implementation Guideline: To avoid overloading issues, use subprograms with distinct names.

8.3.f Packages

Semantic Ramifications

Because the declarations in a package specification are always visible in the package body (LRM 7.1), if an identifier, X, is used in a package body but not declared there, and an X is declared in the package specification, then the package specification's declaration of X is visible in preference to any declaration of X outside the package body that would otherwise have been visible. For example:

```
package OUTER is
     package P is
          X: INTEGER;
     end P;
end OUTER;

package body OUTER is
     X: FLOAT;
     package body P is
          -- reference to X is to INTEGER X, not FLOAT X.
     begin
          null;
     end P;
begin
     null;
end OUTER;
```

This example reflects the rule cited above in 8.0 that identifiers declared in a package specification P are examined before identifiers declared in a context statically enclosing package body P.

Declarations in a package body are considered part of the package specification's declaration list, and hence, redeclaration of an identifier is prohibited in the package body.

Compile-time Constraints

1. Except for subprograms promised in the package specification, an identifier declared in a package specification must not be redeclared (as opposed to overloaded) in the corresponding package body, even if the redeclaration is identical. Hence, a declaration I: INTEGER cannot appear in both the package specification and body.

2. Identifiers used as labels in a package body must not be identical with a directly visible identifier declared in the package specification.

Test Objectives and Design Guidelines

1. Check that inside a nested package body, an attempt to reference an identifier declared in the corresponding package specification is successful even if the same identifier is declared in the outer package body or the outer package specification. (This checks that an implementation looks in the package specification before looking in the statically enclosing context to decide on the meaning of an identifier.)

2. Check that an identifier declared in the package specification cannot be redeclared in the package body. Check for both identical and non-identical redeclarations.

   Implementation Guideline: Don't use subprogram or enumeration identifiers.

   Also check that a label defined in a package body cannot be identical to an identifier declared in the corresponding package specification or body.

3. Check that if a package body is nested inside a package body, the inner package body can contain a label identical to a label in the outer package body or to an identifier declared in the outer package body or its specification.

8.3.g Selected component naming

Semantic Ramifications

Given a name having the form of a selected component, e.g., R.I, if R is a visible package, task, block, loop, or subprogram name, then I must be a subprogram, object, type, subtype, entry, package, task, or exception declared immediately within R. "Immediately" means that R is the innermost unit enclosing the declaration of I.

An enclosing declaration of R can be hidden by an inner declaration:

```
procedure R is
     type T is record
          I: INTEGER;
     end record;
     R: T;
     I: FLOAT;
     J: INTEGER;
begin
... R.I ...      -- refers to component I of record R
... R.J ...      -- illegal; record R has no J component
```

In this case, record R hides the procedure R identifier. Hence, R.x can only refer to a component of record R. R.x is illegal if x is not a component of record R.

There is an interaction here with linear elaboration:

```
procedure R is
     I: FLOAT := 0.0;
     J: FLOAT := R.I;       -- value is 0.0
     type T is record
          I: INTEGER;
          K: INTEGER;
     end record;
     R: T := (2, 2);
     L: INTEGER := R.I;       -- has value 2
     M: FLOAT := I;           -- value 0.0
```

The procedure R identifier is not hidden until after the beginning of the variable R declaration. Hence, subsequent references to R.I are to component I of record R. Outside of the record definition for type T, references to the name I (as opposed to identifier I; see LRM 4.1) are to the floating point variable I, since this I is directly visible outside the record definition (and inside procedure R).

Hiding of identifier R occurs as soon as a new declaration of R is encountered, e.g.,

```
procedure R is
     I: constant INTEGER := 4;
     type R is record
          J: INTEGER := I;      -- R.I would be illegal
          K: FLOAT digits I;    -- R.I also illegal
     end record;
```

The selected component form of naming can only be used to access identifiers whose declarations are in scope. Outside of a procedure, selected component notation cannot be used to access any entities declared inside the procedure, including formal parameter names, e.g.,

```
package body R is
     I: FLOAT;
     function R(I: INTEGER) return INTEGER is
          J: INTEGER;
     begin
     ...
     end R;
     function J return INTEGER is begin ... end J;
begin
     -- references to R.I and R.J are illegal
```

The cited references are illegal because the scopes of INTEGER I and J do not extend outside of function R. Since the declaration of function R hides the package R declaration, references to R.I and R.J can only be attempts to reference components of function R. Since no such components are in scope, the references are illegal.

If "package body" is replaced by "procedure", the declaration of function R does not hide the enclosing procedure declaration. Since procedure R is referenceable in this case, the R in R.I is ambiguous. The LRM states that R.X is therefore (see Gaps) illegal for all X.

Selected component notation cannot be used to identify a formal parameter in a named parameter association, since formal parameter names in this context are restricted to the syntax of identifiers (LRM 6.4). However, within the subprogram, selected component notation is permitted.

Components of private types cannot be accessed using selected component notation except within the private part of a package and within the corresponding package body, since the scope of such components extends only over the private part and the package body.

Note that loop parameters and labels can be referenced with selected component notation. (Ramifications to be further explored.)

Compile-time Constraints

1. When selecting an entity declared immediately within a subprogram, package, task, block, or loop, for a selected component name of the form r.I to be valid (where r is either an identifier, or a valid selected component name):

    a. if r is an identifier, the declaration of r must be directly visible;

    b. the declaration of the entity referred to by r must be the innermost declaration enclosing the declaration of I;

    c. The declaration of I must precede the reference to I;

    d. if r is a subprogram, r.I must be enclosed by the body of r.

    e. if r is a package,

        1. r.I must be enclosed by the specification or body of r; or

        2. I must be declared in the visible part of r's specification (i.e., if I is not declared solely in the private part or package body the reference to r.I can appear outside package r).

    f. r must be unambiguous, even for an overloaded name or identifier.

Test Objectives and Design Guidelines

1. Check that enumeration literals can be referenced using selected component notation.

2. Check that loop parameters can be referenced using selected component notation.

Implementation Guideline: The attempt to reference a loop parameter must be made inside the loop, since the scope of the loop parameter does not extend outside the loop.

3. Check that labels can be referenced with selected component notation.

4. Check that formal parameters, private types and constants, type names, subtype names, subprogram names, exception names, entry names, and names introduced by a renaming declaration can be accessed using selected component notation.

5. Check that parameters and local variables of a subprogram cannot be accessed from outside the subprogram using selected component notation.

6. Check that entities declared in a package body or private part of a package specification cannot be accessed from outside the package.

Gaps

1. Our interpretation of the example with R.I in which package body R is replaced with procedure R (we will call this the "modified example") is not clearly supported by the LRM. Section 4.1.3(e) states that "if there is more than one visible ENCLOSING overloaded subprogram of the given name, the selected component is ambiguous, independently of the identifier" [emphasis supplied]. However, in our example, procedure R encloses the use of R.I, but function R does not. Hence, the rule in 4.1.3(e) does not apply to the example. The fourth paragraph of Section 8.3 states that "The name of an entity declared immediately within a subprogram ... can ALWAYS be written as a selected component within this unit." However, the next sentence goes on to say "The name of the unit ... [must be] UNAMBIGUOUS ...", implying (probably) that if the name is ambiguous (as R is in our modified example), R.I is an illegal selected component name. This contradicts the assertion in the preceding sentence that selected component notation can always be used. One might argue that the error in the second cited sentence (the use of "always") and the statement in 4.1.3 combine to show an intent to permit R.I as a legal reference in our modified example. On the other hand, one could argue that the third cited sentence clearly states that R.I is illegal since R is overloaded, and hence, ambiguous; 4.1.3 merely failed to mention this case. This is the interpretation we have taken, but the wording of the LRM should be revised to remove this confusion.

8.3.h Blocks

8.3.i Access Types

Semantic Ramifications

Incomplete type declarations, e.g., type CAR;, introduce a type name that can be used in subsequent access type declarations, but the name cannot be

used in an object declaration or in component selection until the full declaration has been given.

8.4 Use Clauses

Semantic Ramifications

Interactions with body stubs are discussed in 10.2.

The statement that the effect of a use clause "takes place only on the completion of (its) elaboration" means that the names used to access later packages in the list are not affected by the appearance of earlier packages on the list. For example, given a nested set of packages:

```
package P1 is
   package P2 is
      package P3 is
         ...
      end P3;
      ...
   end P2;
   ...
end P1;
```

a use declaration for gaining direct visibility of all entities in all three packages could be written as:

```
use P1, P1.P2, P1.P2.P3;
```

The form

```
use P1, P2, P3;
```

is not equivalent, since P2 is not directly visible at the place where the use clause appears. However, a succession of use clauses does achieve this effect:

```
use P1; use P2; use P3; -- legal
```

The first use clause makes P2 directly visible (assuming no naming conflicts, for the moment); then the second clause makes P3 directly visible.

Note also that in the preceding example, use P3 can appear immediately after end P3, making P3's declarations directly visible within P2 after the use_clause.

The third paragraph of the LRM mentions "identifiers (and entities)". Entities that are not referenced with identifiers are character literals and operator symbols.

It should be noted that the effect of a use clause is not transitive,

i.e., since use clauses only affect the visibility of entities declared in the named packages, and since a use clause does not declare any entities (it only makes them visible), then:

```
package P is
   A: ... ;
end P;

package R is
   use P;         -- A is directly visible now
   ...
end R;

package S is
   use R;         -- A is not visible in S
   use P;         -- now A is visible
   ...
end S;

package body R is
        -- A is visible
```

Although the effect of a use clause in independent package specifications is not transitive, since package bodies behave as extensions of their specifications with respect to visibility (see 7.1), a use clause in a package specification has an effect in the declarative part of the corresponding package body. For example, if A is directly visible after the last declaration in R's specification, it is also directly visible at the beginning of R's body. Note that A will no longer be visible in R after a redeclaration of A in R:

```
package R is
   use P;         -- A is directly visible now
   A: ...         -- P.A is now hidden
```

Similarly, note that in the previous example in which use P3 was inserted after the declaration of P3, any package using P1 and executing use P1,P2 would not obtain direct visibility of the entities declared in P3.

Compile-time Constraints

1. The names appearing in a use clause must designate packages.

Test Objectives and Design Guidelines

1. Check that only packages can be designated in use clauses. In particular, check that task specifications and subprogram specifications cannot be designated.

2. Check that a use clause enclosed in a task, package or subprogram body can use the name of the enclosing unit to form a package name appearing in the use clause.

3. Check that if an identifier appears as an enumeration literal, subprogram identifier, or object identifier in only one of several packages, a use clause naming all the packages makes the identifier directly visible after the use clause is elaborated, but not before.

4. Check that if an identifier, P, is declared as a subprogram identifier with distinct signatures in several packages (S1, S2, ...), as an object in packages O1, O2, ..., and as an enumeration literal in packages E1, E2, ...:

   - a use clause mentioning only packages in the set S makes all the subprograms P directly visible;

   - if the use clause mentions packages in the set S and E, but not O, the subprograms and enumeration literals named P are directly visible;

   - if a later use clause mentions package O, only the enumeration literals named P are subsequently directly visible;

   - if the use clause mentions packages in the set S and O, none of the entities named P are directly visible;

   - if a use clause mentions packages in the sets E and O, only the enumeration literals are made directly visible. BUL if the use clause mentions packages in the set S, E, and O, only the enumeration literals named P are directly visible;

5. Check that the effect of a use clause in a package specification is apparent in the corresponding package body. Check that the effect of a use clause in a package, subprogram, task body, or block does not extend outside the unit containing the clause (see Gaps).

6. Check that if an identifier is directly visible, and is the same of a subprogram or enumeration literal, then a use clause designating a package containing a declaration of the name identifier as an enumeration literal, subprogram identifier, or object identifier does not make these identifiers directly visible.

7. Check that if an overloaded subprogram call can be given an unambiguous interpretation without considering identifiers made directly visible via a use clause, it is given that interpretation, even if consideration of the use clause would have made the call ambiguous.

   Check that a call which has no interpretation if use clauses are ignored can be given the proper interpretation when use clauses are present.

Gaps

1. The LRM does not state explicitly that the effect of a use clause is limited to the scope of the declarative part in which the clause exists.

8.5 Renaming Declarations

Semantic Ramifications

For object renamings, "establishing" the identity of the object being renamed is the same process as establishing the identity of a variable used as an in out or out parameter in a subprogram call (see 6.4.1). In essence, renaming performs a "by reference" binding to the renamed object. For subprogram renamings, "establishing" the identity of the renamed subprogram means performing a kind of overloading resolution on the specified subprogram name. Note however that the matching criteria are different (e.g., names of parameters don't matter and names of subprograms made visible by use clauses are immediately available for matching - there is no tow step matching process as for overload resolution (see LRM 6.6)).

The motivation underlying the requirement that the first form of renaming declaration "express the same constraints as those of the renamed entity" is to ensure that assignments to the new name raise an exception if and only if assignments to the renamed object would raise an exception (since the new name is just "another name for entity"). Given this motivation, one can work out what "express the same constraints" must mean (see G/17) for a renaming declaration of the following form:

```
OBJECT: [constant] R ... ;
X: T renames OBJECT;
```

where OBJECT is designated by an access value or is declared in a parameter, object, or component declaration and where R is a name for a stype indication. The required checks are then as follows. If T is:

- a scalar type, T'FIRST must equal T'LAST and R'FIRST must equal R'LAST;
- a floating point type, then T'DIGITS must equal R'DIGITS;
- a fixed point type, then T'DELTA must equal R'DELTA;
- a constrained array type, then R must be constrained and for all indexes, i, T'FIRST(i) must equal R'FIRST(i) and T'LAST(i) must equal R'LAST(i);
- an unconstrained array type, then R must also be unconstrained (only possible when renaming formal parameters; see G/18).
- an unconstrained record, private, or limited private type, R must also be unconstrained (i.e., R'CONSTRAINED must be FALSE);
- a constrained record, private, or limited private type, R must also be constrained (i.e., R'CONSTRAINED must be TRUE) and the renamed object must satisfy T's discriminant constraints (i.e., the constraint values must be the same);

- an unconstrained access type, R must also be unconstrained;

- a constrained access type, R must also be constrained and must have the same constraint values.

A record or array object designated by an access object can only be renamed with a constrained record type, since the constraints for such an object cannot be changed (see 3.8 and G/9). For example,

```
type PERSON_NAME is access PERSON;
JOHN : PERSON_NAME := new PERSON(M);
JEAN : PERSON renames JOHN.all; -- illegal
```

Note that JOHN.all'CONSTRAINED is TRUE (see 3.7.2) but JEAN'CONSTRAINED would have the value FALSE. The renaming is illegal because the type constraints for JEAN imply JEAN can be assigned a PERSON(F) value, but such an assignment would violate the constraint for the renamed object, JOHN.all, which is required to always hold a PERSON(M) value (see 3.8), since it was initialized with a PERSON(M) value.

Note that an unconstrained array type or an unconstrained record or private type without default discriminants can be used (see G/18) in the first form of renaming declaration only to rename unconstrained formal parameters, since this is the only context permitting such unconstrained types, other than constant declarations. (We assume that when attempting to rename a constant declared with an unconstrained type, the renaming declaration must specify a constrained type whose constraints are equal to those of the constant object. After all, renaming applies to objects and is unaffected by how the object is declared unless the method of declaration affects the constraints associated with the object, as is the case for formal parameters.)

Note that there are two ways in which a component can depend on the value of a discriminant: 1) it can be a component of a variant part; or 2) it can be a component of an array whose bounds depend on a discriminant, e.g.:

```
type T(L: INTEGER range 1 .. 100 := 0) is
     record
          A: array (1..L) of CHARACTER;
     end record;
DATA: T;
CONS: T(50);
```

Although no component of DATA.A can be renamed, any component of CONS.A can be renamed, since CONS is constrained. In addition, suppose we have a procedure:

```
procedure P (X: T; Y: in out T);
```

Any component of X.DATA can be renamed, since X acts as a constant and hence its discriminant values are fixed. A component of Y.DATA can be renamed if Y'CONSTRAINED is true, which will be the case if the actual parameter is CONS. An attempt to rename Y.DATA components will fail if the actual parameter is DATA. Since this renaming failure can only be determined at run time, presumably CONSTRAINT_ERROR is raised (see G/15).

If the renamed object is a constant object, an in formal parameter, a loop parameter, or a record discriminant, the new name cannot be used in an assignment context since "the newly declared identifier is constant if the renamed entity is."

The renamed entity cannot be the name of an entry family (although it can be the name of an entry in a family) since an entry family is not an object, exception, package, task, or subprogram, and these are the only entities that can be renamed. Similarly, it cannot be the name of a number, since a number is not an object (see 3.2), nor can it be a label, block identifier, or loop identifier.

The new identifier introduced by a renaming declaration can be used in selected component notation as a replacement for the renamed entity, e.g., if A is an array of records and A(I).B.C is the name of an INTEGER component, then:

```
A_I_B : INTEGER renames A(I).B;
```

means that A_I_B.C is a valid name for that component. Note that the value of I is determined at the time the renaming declaration is elaborated, and subsequent changes to the value of I have no effect on which component is associated with A_I_B, e.g., if I = 5 when the renaming declaration is elaborated, then:

```
I := 7;
... A_I_B.C     -- equivalent to A(5).B.C
```

Note that if the renamed entity is in package P and the renaming declaration is also in package P, then P.A_I_B.C is equivalent to P.A(I).B.C. However, renaming must not be considered a macro definition. Consider:

```
package P is
   type T is record ... end record;
   A: array (1..10) of T;
end P;

package Q is
   use P;
   A_I_B: INTEGER renames A(I).B;
end Q;
```

Note that P.A(I).B.C is a valid name in Q, but P.A_I_B.C is not allowed in Q, since there is no name A_I_B declared in P. However, Q.A_I_B.C would be valid, since A_I_B is a name declared in Q; the fact that this name designates an object in P is irrelevant.

Note that there is no restriction forbidding the renaming of generic formal parameters or their components.

Note that a slice is a name, and hence, slices can be renamed as arrays.

Note that there is no restriction on the use of the first form of renaming declaration to rename task objects declared with a task type name, e.g.,

```
task type T;
T1: T;
U: T renames T1;
```

Note that the renaming

```
task U renames T1;
```

could also be used.

No constraints are stated on what can be renamed as a task. We assume that the intent is to permit renaming of task objects (either objects declared with task type names or single tasks declared with a task specification). We suspect it was not intended that a renaming declaration be used for task types (see G/14), e.g., the following declaration is probably not supposed to be permitted:

```
task U_TYPE renames T;
```

where T is the name of a task type.

A renamed task is not activated as a result of the renaming, since renaming does not create a new object, but just a new name for an existing object. The new task name can be used to access entries of the renamed tasks, however.

If a package is nested inside another package, the inner package can be renamed in an outer scope, e.g.,

```
package P is
   package Q is
      package R is
         ...
      end R;
   end Q;
   package NEW_R renames Q.R;
end P;
```

Now separately compiled units can access elements of Q.R as though they were contained in package NEW_R declared at the outermost level of P. Note also that use NEW_R has an effect equivalent to use Q.R.

A renaming declaration in a package body or the private part of a package specification cannot be used to satisfy the requirement for a complete declaration of a subprogram body or constant. For example, the following renaming declarations are illegal:

```
package P is
   procedure PROC;
   type PRIV is private;
   X: constant PRIV;
private
   type PRIV is INTEGER;
   XX: constant PRIV := 0;
   X: PRIV renames XX;  -- illegal
end P;

with R;
package body P is
   procedure PROC renames R.PROC;        -- illegal;
end P;
```

These declarations are illegal because a renaming declaration does not introduce a new constant or subprogram -- it merely renames an existing one. Moreover, the rules for subprograms declared in packages state that a subprogram body must appear in the corresponding package body (see 7.1 and 7.3), and a renaming declaration does not define a subprogram body. Similarly, private constants must be redeclared "in full" (LRM 7.4), and a renaming declaration is not in any sense a redeclaration in full.

It is unclear whether the following kind of renaming and use of the new name is intended to be permitted:

```
package P is
    procedure Q(A: INTEGER);
    procedure Q(B: FLOAT);
private
    procedure Q1(A: INTEGER) renames Q;
    procedure Q2(B: FLOAT) renames Q;
end P;

package body P is
    procedure Q1 (A: INTEGER) is ... end Q1;
    procedure Q2 (B: FLOAT) is .... end Q2;
end P;
```

Sine Q1 and Q2 "can be used as the name of [the renamed] entity" after the renaming declarations in P, do the declarations of the bodies Q1 and Q2 satisfy the need to specify Q's bodies (6.3)? Can an INLINE pragma (6.3) be given for Q1 and Q2? Can INLINE be specified just for Q1? The answers are not clear, but the above capability would be useful if Q1 and Q2 were to be declared as subunits (see 10.2.S) or if an INTERFACE pragma (see 13.9) is to be specified for Q1 and Q2 (because the language in which the bodies of Q are to be written does not support overloading). At present, the above example does not seem to be explicitly forbidden, but we are not sure whether the intent was to permit such uses of renamed entities (see G/19).

Since the SUPPRESS pragma (see 11.7) is object-oriented, whether a SUPPRESS pragma gives a new name for an object or its original name, the

effect is the same. In particular, regardless of the name used in the pragma and regardless of what name is used to access the object named in the pragma, the effect of the pragma will be the same.

Compile-time Constraints

1. The name appearing after renames must not be the name of a statement label, the name of a block or loop identifier, the name of an entry family, the name of a number, the name of an enumeration literal, or the name of a function_call (see G/12, 13).

2. Only a variable or constant (including formal parameters) can be renamed in the first form of renaming declaration.

3. The base type of the type_mark and of the renamed entity must be the same (see G/7).

4. If the type_mark is an unconstrained array, access, record, private, or limited private type, the renamed object must also have been declared with an unconstrained type mark (of the same base type) and must not be an object designated by an access object (see G/9, 18).

5. Only the name of an exception condition can be renamed as an exception (see Gaps).

6. Only the name of a package can be renamed as a package (see Gaps).

7. Only the name of a task object can be renamed as a task (see G/14, 4).

8. Only the name of a function, procedure, entry, or attribute (?; see G/1) can be renamed in the last form of renaming declaration.

9. A function cannot be renamed as a procedure, and vice versa (see G/2).

10. An entry can only be renamed as a procedure.

11. A value returning attribute can only be renamed as a function having the appropriate number of parameters and returning the appropriate type of value (see G/1).

12. If the renamed entity is a loop parameter, discriminant, constant object or component of a constant object, in formal parameter, or a renamed constant, the new identifier cannot be used as the target of an assignment statement, or as an actual in out or out parameter.

13. The number of parameters in the subprogram_specification must equal the number of parameters associated with the renamed subprogram.

14. The base types of corresponding parameters (and the result type) must be the same as when renaming subprograms (see G/11).

15. If one formal parameter (result type) is declared with a constrained

composite type, its corresponding formal parameter (result type) must not be declared with an unconstrained type (see G/3).

16. The modes of the corresponding formal parameters must be identical.

Exceptions

1. CONSTRAINT_ERROR is raised for the first form of declaration if:

   - the range of values associated with a scalar type_mark is not identical with the range of values associated with the subtype of the renamed entity;

   - the 'DIGITS attributes of the type_mark and renamed object are not equal;

   - the 'DELTA attributes of the type_mark and renamed object are not equal;

   - if the type_mark is a constrained array type, the index bounds of the type_mark and the renamed object are not equal;

   - if the type_mark is a constrained record, private, or limited private type, the discriminant values associated with the type_mark and the renamed object's subtype are not equal.

   - if the type_mark is a constrained access type, the index or discriminant constraint values associated with the type_mark and the subtype of the renamed access object (not the designated object) are not equal;

   (The first three checks (see G/20) are suppressed by RANGE_CHECK applied to the renamed object or its type; the last three are suppressed by INDEX_CHECK applied to the renamed object, its type, or (for a given discriminant), the type of the discriminant or the name of the discriminant component.)

2. CONSTRAINT_ERROR is raised if the renamed object is a component of a variant part or a component of an array with a bound specified by a discriminant value and the 'CONSTRAINED attribute for the object containing the renamed component is FALSE (see G/15). (It is unclear what check name is used to suppress this exception check; see G/16.)

Gaps

1. Whether attributes can be renamed is unclear in the current LRM. Some attributes with parameters (see 3.5.5) are said to be functions, and so, presumably, the functional form of renaming must be available for them. Other attributes with parameters (see 3.6.2) are not necessarily considered functions, e.g., presumably A'RANGE(n) must be renamed with a subtype declaration. Parameterless attributes are said to be values (see 4.1.4), leaving open the question of whether they can be renamed at all.

It is unclear whether all attributes returning values are to be considered functions that can be renamed using a subprogram specification. For example, consider renaming the LENGTH attribute function for a type:

```
function L (I: INTEGER := 1) return INTEGER renames T'LENGTH;
-- L(2) is equivalent to T'LENGTH(2)
-- L() is equivalent to T'LENGTH or T'LENGTH(1)
```

The argument to T'LENGTH must be static. Must the argument to L also be static?

2. The rules for matching the subprogram specification and the entity being renamed do not state whether a function can be renamed as a procedure (presumably not).

3. When matching the constraints for a subprogram renaming, it is not clear whether the rule requiring that the constraints for formal parameters and result be the "same" means they must be lexically the same (except for possible variation in the form of names used) or if the values of various constraints must be identical. Since the subprogram specification is elaborated in the renaming declaration, presumably the intent is to require that the subprogram specification must "express" the same constraints as the renamed entity, i.e., the values must be the same, but not necessarily the form expressing these values. Similarly corresponding array, record, or access parameters must both be constrained or both be unconstrained.

4. It is not stated explicitly that the forms for renaming exceptions, packages, and tasks can only be used to rename these entities, respectively.

5. It is unclear whether the identifier introduced by a renaming declaration can be used in named association for aggregates since "a renaming declaration declares another name for an entity." We suspect that such usage of new names is forbidden.

6. It is not stated explictly that the base type of the type_mark and the base type of the renamed object must be the same.

7. It is unclear whether the identifier introduced by a number declaration can be renamed. We have assumed that it cannot be, since such an identifier does not denote an object; it is merely the name for a number (see 3.2).

8. The LRM does not state what exception is raised if the constraints given in an object declaration do not "express" the same constraints as the renamed object. We have assumed CONSTRAINT_ERROR is raised except under situations where equality of constraint values is not at issue, e.g., when the type mark is an unconstrained record type, but the renamed object is constrained.

9. The LRM does not clearly imply that an object designated by an access value can only be renamed as a constrained record or private type.

10. The matching rules for renamed subprograms do not require that the number of parameters in the subprogram specification and the renamed subprogram be the same. This appears to be an oversight, since the only situation in which some meaningful interpretation can be generated is when the renamed subprogram has one or more parameters with default values at the end of its list, and the subprogram specification omits, say, the last parameter. It could be implied that the default value of the missing parameter is to be used in all calls of the renamed subprogram. However, since defaults are ignored in the matching, this is probably not the intent.

11. The matching rules state that the values of parameter (and result) constraints are taken into account, and later, it is stated that a renaming declaration is illegal if more than one subprogram "matches" the subprogram specification. This says that constraints are taken into account when deciding whether a renaming is ambiguous. For example consider

```
procedure P(A: INTEGER range 1..10);
procedure P(B: INTEGER range 1..20 := 1);
procedure Q(A: INTEGER range 1..20) renames P;
```

The two overloadings of P are acceptable because one has a default parameter and the other does not (see 6.6). A strict reading of 8.5 says that Q renames the second declaration of P, since that is the only declaration with matching constraints. Note that parameter names and the presence or absence of default values are not considered in defining the matching. However, if the ranges on these parameter values are not defined with static expressions, the existence of an ambiguity cannot be determined at compile time. We suspect that the intention was to require that the determination of ambiguity be performed at compile time, and hence, the matching rule should not have mentioned constraints. The specification should have said that if a match is made (ignoring constraints), CONSTRAINT_ERROR is raised if the constraint values are not the same and the program is illegal if corresponding parameters in the subprogram specification and renamed subprogram are not both either constrained or unconstrained.

12. It is unclear whether the result of a function call can be renamed as an object, since a function call returns values (6.5), not objects. However, the result of a function call can be subscripted or selected (see 4.1), in which case, the result does have some of the properties of an object. We have assumed function call results cannot be renamed.

13. Similarly, can an enumeration literal be renamed as an object? Once again, enumeration literals are defined as values, not objects, so it appears likely that such a renaming is not permitted.

14. It is unclear whether the task renaming form can be used to rename task types as well as task objects. Since there are no restrictions forbidding the use of this form to rename task types, one might assume it is permitted. However, there are also no restrictions on the use of subtype declarations to rename task types. Since subtype declarations are used to

rename types other than task types, it might be reasonable to assume that subtype declarations are to be used for task types as well; there are no restrictions forbidding such declarations. Hence, at present there seem to be two ways of renaming task types -- by renaming and by subtype declaration. We suspect that the intent was to provide only one method of renaming task types and that subtype declarations were the intended method.

15. It is not stated what exception condition is raised if an attempt is made to rename a component of a formal parameter when the component's existence depends on a discriminant and 'CONSTRAINED is FALSE. The implications of the phrase "whose existence depends on the value of a discriminant" is also unclear in cases like the following:

```
type T (L: INTEGER range 5..100) is
     record
          DATA: array (1..L) of INTEGER;
     end record;
```

It could be argued that the existence of DATA components 1..5 does not depend on the value of L, since L can never have a value less than six. However, since the range constraints for L can, in general, be non-static, we have assumed that either all or none of the DATA components can be renamed.

16. It is unclear what CHECK name (see 11.7) is to be used to suppress the constraint checks required in renaming declarations. Presumably RANGE_CHECK applies to checks for equality of constraint values, but DISCRIMINANT_CHECK might be used to suppress the checks to see if the existence of a component depends on the value of a discriminant.

17. When renaming objects, we have assumed that "express the same constraints" means the values of constraints must be identical, not that the forms in which the constraints are specified must be identical. In general, the phrase "express the same constraints" requires more explanation.

18. We have assumed it is possble to rename unconstrained formal parameters or components of such parameters and that the new name takes on the same constraint values and properties as the normal parameter has for a particular subprogram invocation. It would be reassuring if the LRM contained a note indicating that this kind of renaming was intended to be permitted.

19. It is unclear whether renaming a subprogram specification in a package specification means a body for the renamed subprogram can be given using the new name.

20. The definition of RANGE_CHECK in 11.7 does not clearly cover the case of checking whether <u>digits</u> or <u>delta</u> values are identical in renaming declarations.

Test Objectives and Design Guidelines

Renaming Objects

1. Check that renaming is forbidden for names of numbers.

   Check that a subtype indication cannot be used in a renaming declaration.

   Implementation Guideline: The subtype indication should contain constraints equal to those of the renamed object. Not all forms of constraint need be checked.

   Check that a constrained record, private, or access type cannot be used to rename an object declared without a constraint, and vice versa.

   Implementation Guideline: The access type should designate both array and record objects.

   Check that a component of a variant part cannot be renamed in a record whose discriminant values are not constrained (see also T/).

   Check that neither a component nor a slice of an array having a bound defined by a discriminant can be renamed in a record whose discriminant values are not constrained (see also T/13).

   Check than an array (or record) object designated by an access type cannot be renamed with an unconstrained array (or record) type (see also T/16).

   Check that an enumeration literal, exception, package, block name, loop name, entry, or subprogram cannot be renamed as an object.

   Check that the value of a function call cannot be renamed (see Gaps).

2. Check that if M renames a constant, in formal parameter, loop parameter, component of a constant object, renamed constant, or record discriminant, M cannot be assigned to or used as an actual in out or out parameter.

   Implementation Guideline: Rename not only a scalar object, but also a record and array object and attempt to assign to a component of the renamed record or array.

3. Check that an unconstrained array, record, private, or limited private type mark can be used in a renaming declaration to rename an unconstrained formal parameter, and that (for objects with discriminants) the new name has an appropriate 'CONSTRAINED value, takes on the constraint values of the formal parameter, and can be assigned to in a way that changes the formal parameter's constraints if such assignments are permitted to the actual parameter.

   Implementation Guideline: Use declarations in which new names, T1 and T2, are introduced (via subtype declarations) for the base type, T. Then check that the following combinations are accepted, where FP is the

type_mark used in the formal parameter declaration and TM is the type_mark used in the renaming declaration: FP = T and TM = T1; FP = T1 and TM = T; FP = T1 and TM = T2; FP = T and TM = T.

4. Check that when renaming objects of an integer or enumeration type, the 'FIRST and 'LAST attributes of the renamed object's subtype and the new name's type_mark must be equal, and if not equal CONSTRAINT_ERROR is raised.

   Implementation Guideline: Use both static and non-static expressions in specifying the required range constraints.

5. For an implementation that supports more than one predefined integer type, check that the base types of the new name and renamed object cannot be different integer types, even if the range constraints are identical

   Implementation Guideline: For example, use declarations like the following:

```
subtype SHORT is  INTEGER range 0..SHORT_INTEGER'LAST;
X: SHORT_INTEGER range 0..SHORT_INTEGER'LAST;
Y: SHORT renames X;      -- illegal; base types not identical
```

   Repeat this check for predefined floating point types as well.

6. Check that when renaming objects of a floating point type, the 'DIGITS, 'FIRST, and 'LAST attributes of the renamed object's subtype and the new name's type_mark must be equal, and that if unequal, CONSTRAINT_ERROR is raised.

   Implementation Guideline: Use both static and non-static expressions in specifying the required range constraints. Use a digit value in the type_mark that is less than the required digits value to maximize the chance of detecting an error.

7. Check that when renaming objects of a fixed point type, the 'DELTA, 'FIRST, and 'LAST attributes of the renamed object's subtype and the new name's type_mark must be equal; if any one of these is unequal, check that CONSTRAINT_ERROR is raised.

   Implementation Guideline: Use both static and non-static expressions in specifying the needed range constraints. Use a delta value in the type_mark that is greater than the required delta value to maximize the chance of detecting an error.

8. Check that when renaming an array object (including a slice and a constrained formal array parameter), the index ranges associated with the type_mark in the renaming declaration must specify the same index bounds values that are associated with the renamed object. Check that CONSTRAINT_ERROR is raised if any bound is not the same.

   Implementation Guideline: Try both single and multi-dimensional arrays with static and non-static index bounds.

Check that if assignments are made to the new name, the renamed object is modified, and vice versa.

9. Check that when renaming a record, private, or limited private object (other than a formal parameter) declared with an unconstrained type, the new name must also be declared with an unconstrained type.

<u>Implementation</u> <u>Guideline</u>: Try a declaration using a constrained type_mark where the constraint values match those of the renamed object's current value.

Check that if assignments are made to the new name, the renamed object is modified and that the assignment can change the discriminants associated with the renamed object. Check also that assignments to the renamed object change the value associated with the new name.

10. Check that when renaming a record, private, or limited private object declared with a constrained type, the new name must also be declared with a constrained type having the same discriminant values. Check that CONSTRAINT_ERROR is raised if any constraint values are not equal.

<u>Implementation</u> <u>Guideline</u>: Use both static and non-static specifications of discriminant values.

<u>Implementation</u> <u>Guideline</u>: Attempt to rename uninitialized record objects.

<u>Implementation</u> <u>Guideline</u>: Use types with and without default discriminant values.

11. Check that an access object declared with an unconstrained access type can be renamed with an unconstrained access type_mark. Check that assignments to the new name change the value of the renamed object (and vice versa), and that these assignments can designate objects with different discriminant values.

Check that an unconstrained access formal parameter can be renamed and that assignments to the new name are subject to the same constraints as assignments to the formal parameter.

12. Check that an object declared with a constrained access type can only be renamed with a type_mark having identical constraints. Check that if the constraint values are not equal, CONSTRAINT_ERROR is raised.

<u>Implementation</u> <u>Guideline</u>: Use both static and non-static constraint values.

13. Check that the renamed object can be a component of an array or slice.

<u>Implementation</u> <u>Guideline</u>: The array components should be of a scalar, array, record, private, limited private, and access type, with constraint values both statically and non-statically defined.

Implementation Guideline: The array component should be specified with both static and non-static index values. When non-static values are used, check that subsequent changes to variables in the expression do not change the component that is accessed.

Check that assignments to the array as a whole change the value of the renamed component or slice when the new name is used.

Check that if the array is a record component and has a bound depending on a discriminant, a component of the array can be renamed if the record object was declared with discriminant constraints, or if the record is designated by an access object, or if the record is associated with a formal parameter P, and P'CONSTRAINED is TRUE. In this last case, also check that CONSTRAINT_ERROR is raised if P'CONSTRAINED is FALSE.

Implementation Guideline: Note there are three ways P'CONSTRAINED can be true if P is a formal parameter: 1) P is declared as a constrained formal parameter, 2) P is unconstrained, but the actual parameter is constrained, and 3) P is an unconstrained <u>in</u> parameter (even if the actual parameter is unconstrained). All three situations should be checked.

14. Check that the renamed object can be a component of a record.

Implementation Guideline: Try renaming components of records whose existence does not depend on the value of a discriminant and for records declared without a discriminant constraint (including records associated with formal parameters).

Check that a component of a variant part can be renamed if the record containing the component has a true 'CONSTRAINED attribute.

Implementation Guideline: 'CONSTRAINED is true if the record object was declared with a discriminant constraint, or if it is designated by an access value, or if the record is the value of an unconstrained formal parameter and the actual parameter is constrained, or the formal parameter is an <u>in</u> parameter.

15. Check that a new name for a record (or array) component cannot be used when specifying an aggregate for the record (or array) using named component associations.

16. Check that an object designated by an access value, e.g., PTR.A, can be renamed and that access to the designated object is permitted even after there are no access values designating it.

Check that if the object is an array or record, it can be renamed with a constrained array or record type, and that the values of the constraints associated with the type_mark must equal the constraint values associated with the object, or else CONSTRAINT_ERROR is raised.

Implementation Guideline: Try to rename uninitialized as well as initialized record and array objects.

Implementation Guideline: Use both static and non-static constraints when defining the type_mark used in the renaming declaration.

Implementation Guideline: Use both constrained and unconstrained access types to generate the object being renamed.

17. Check that if the new name for an array or record component itself denotes an array or record, the new name can be used in forming an indexed_component name or a selected_component name.

18. Check that if an object in another package is renamed, the new name does not serve as a macro replacement for the old name (see the P.A_I_B example in the Semantic Ramifications).

19. Check that a task object can be renamed using the first form of renaming and an explicit task type name.

Check that a task object declared with a task specification cannot be renamed (since the task name is not also its type name), e.g.,

```
task T;
U: renames T;   -- illegal
```

Renaming exceptions

30. Check that an exception cannot be renamed as an object, package, task, task name, a loop name, entry or subprogram.

31. Check that both user-defined and pre-defined exceptions can be renamed.

Check that exceptions declared in packages can be renamed outside the package.

Implementation Guideline: Use the new name both in raise statements and in exception handlers, and check that a handler using the new name will catch an exception raised using the old name, and vice versa.

32. Check that the T'FAILURE exception can be renamed (e.g., as FAILURE), and that when the new exception name is raised, T'FAILURE is raised in the appropriate task.

Renaming packages

40. Check that an object, task, exception, entry, subprogram, block name, or loop name cannot be renamed as a package.

41. Check that a nested package can be renamed outside the unit containing the package, and that the new name can be used in selected component naming or a USE clause to gain visibility of the inner package's components.

Implementation Guideline: Check these effects both within and across separately compiled packages.

42. Check that a generic package can be renamed and that the new name can be instantiated.

Renaming tasks

50. Check that an object, package, exception, entry, subprogram, block name, or loop name cannot be renamed as a task.

Check that a task type cannot be renamed as a task (see Gaps).

51. Check that a task object declared with a task type name can be renamed as a task.

Implementation Guideline: Check using arrays and records of tasks, tasks designated with access objects, and tasks declared in object declarations.

Check that renaming a task object does not cause activation of the renamed task.

Renaming subprograms

60. Check that an object, package, exception, block name, loop name, or task cannot be renamed as a subprogram.

Check that a function cannot be renamed as a procedure (and vice versa) (see Gaps).

Check that an entry cannot be renamed as a function.

Check that an attempted renaming is illegal if the subprogram specification and the subprogram being renamed differ in only one of the following ways:

- The number of parameters in the specification is greater than the number of parameters in the named subprogram;
- the number of parameters in the specification is fewer than the number of parameters in the named subprogram (even if the omitted parameters have default values and appear at the end of the parameter list);
- the modes of one of the parameters do not match (try all 6 mismatches);
- the base types of one of the parameters do not match; (try both derived and non-derived types for the parameters);
- for functions, the result base types are different;
- there is more than one subprogram that matches the subprogram specification and name;

<u>Implementation Guideline</u>: The allowable differences to be checked are: 1) two subprograms identical except for their formal parameter names (note: these must be declared in nested scopes; see 6.6); 2) two subprograms differing in the presence or absence of default values for a parameter position; 3) two subprograms differing in their default values for a given position. The subprogram specification in the renaming declaration should be identical to the specification for one of the ambiguous subprograms.

- one parameter is a constrained record, private, array, or access type and the corresponding parameter in the renamed subprogram is not, and vice versa.

Check that renaming cannot be used to introduce a subprogram intended to satisfy a declaration given in a package specification.

61. Check that in the absence of overloading, the new subprogram is considered to match the named subprogram even if the specification for the new name and the existing name differ in only one of the following aspects:

    - presence/absence of default values for a parameter;
    - the value of a parameter's default value;
    - the name of a formal parameter.

    Check that when the new name is called, the appropriate default parameters are permitted, and any new default values are supplied to the renamed procedure.

    Check that named parameter association for the new subprogram name can be used.

62. Check that if renaming introduces an overloading for a subprogram name, the names of the formal parameters and use of default parameter values serve to indicate which subprogram is to be invoked.

63. Check that CONSTRAINT_ERROR is raised if the constraint values for a parameter in a subprogram specification do not equal the values of the corresponding parameter in the renamed subprogram.

64. Check that a generic subprogram can be renamed and instantiated using the new name.

65. Check that if a subprogram name is made visible by a <u>use</u> clause, both it and any names directly visible in the absence of a use clause are candidates for renaming.

    <u>Implementation Guideline</u>: Two tests are needed. In one, the renaming should be ambiguous only if the use clause is taken into account. In the second, the renamed subprogram should be one that is directly visible only by virtue of a <u>use</u> clause's effect.

66. Check that an entry or a member of an entry family can be renamed as a procedure.

Renaming generic parameters

70. Check that any generic formal parameter can be renamed within a generic unit by repeating the above tests for generic parameters.

8.6 Predefined Environment

Semantic Ramifications

Because library units mentioned in with_clauses are implicitly declared in STANDARD (and because only these library units are so declared), these library units can be named as components of STANDARD within the unit being compiled. For example, if L is a unit named in a with_clause, STANDARD.L is also a name for that unit. However, because the names specified in a with_clause are implicitly declared in STANDARD only while a compilation unit is being processed (see G/1), it is not possible to name a library unit M as STANDARD.M if M does not appear in the with_clause associated with the unit being compiled.

Note that if STANDARD is not a library unit (see G/2), a library unit can be given the name STANDARD. If such a unit is a package, it would not replace the declaration of STANDARD; it would merely act as a package named STANDARD that is declared within the predefined STANDARD package. Of course, within such a user-defined library unit, the name STANDARD refers to the user-defined unit, not the predefined package, just as when one declares a package P within package P, the outer package name is hidden within the inner package.

Note that STANDARD is not a reserved word. Hence, within a declarative part, the following declarations are permitted.

```
type T is
     record
          INTEGER: FLOAT;
     end record;
STANDARD: T;
```

Now the name STANDARD.INTEGER refers to the component of the record object named STANDARD. Note also that if T is declared within package P, and P is a library unit, STANDARD.P.STANDARD.INTEGER is an illegal name, since 1) the first identifier in a selected component name refers to a directly visible identifier (see LRM 8.3), 2) only the record object STANDARD is directly visible, and 3) there is no P component in the record object.

We assume (see G/3) that while a library unit (not subunit; see LRM 10.1) is being compiled, it is considered to be implicitly declared in STANDARD. If this assumption is correct, it implies that no library unit name can be the same as an identifier declared in STANDARD since the implied declaration of the library unit within STANDARD would then be an illegal redeclaration within

STANDARD (see the last paragraph of LRM 8.3). For example, an attempt to compile a library unit named STRING would be illegal, since it would conflict with the type name STRING declared in STANDARD. (Note that even without our assumption, a library unit named STRING is virtually useless, since it cannot be referenced in a with_clause without introducing an illegal redeclaration in STANDARD; the current wording of the LRM would seem to permit STRING as a main program never appearing in any other unit's with_clause, since the current wording does not explicitly say that while a library unit is being compiled, it is implicitly declared within STANDARD).

Note that operators cannot be declared as library units (see 10.1).

Note that when compiling a subunit, the LRM states (in 10.2) that "visibility is as at the corresponding body stub." Consequently, we assume (see G/4) that the parent library unit is implicitly declared in STANDARD. Hence, within the subunit, unique selected component names starting with STANDARD can be used to designate identifiers in any ancestor unit. For example:

```
with Q;
procedure P is
    I: INTEGER;
    procedure Q is separate;
begin
    ...
end;

separate (P)
procedure Q is ... end Q;
```

Within Q, the name STANDARD.P refers to the parent procedure, the name STANDARD.P.I refers to the variable declared in P, and the name STANDARD.P.Q.x could be used to refer to an identifier declared within Q. Similarly, STANDARD.Q.x refers to an identifier declared in package Q, whereas Q.x refers to an identifier in Q.

A subunit can be named the same as some identifier in STANDARD since subunit names are not implicitly declared in STANDARD.

Compile-time Constraints

1. If the name of a library unit is an identifier, the identifier cannot be identical with a predefined identifier declared in STANDARD (see G/3).

Test Objectives and Design Guidelines

1. Check that the name STANDARD.M is illegal if M does not appear in a with_clause and the name of the library unit being compiled is not M.

   Implementation Guideline: M should be the name of some previously compiled library unit.

Check that a library unit cannot have a name identical to a type, exception, or package declared in STANDARD, namely, BOOLEAN, INTEGER, FLOAT, CHARACTER, ASCII, NATURAL, PRIORITY, STRING, DURATION, CONSTRAINT_ERROR, NUMERIC_ERROR, SELECT_ERROR, STORAGE_ERROR, TASKING_ERROR, or SYSTEM (see also 10.1.T/1), plus any implementation defined numeric type name (e.g., LONG_FLOAT).

2. Check that a library unit can be given the name STANDARD. Try compiling both a STANDARD package (with redefinitions of some predefined types and operators) and a subprogram called STANDARD. Check that the predefined STANDARD package is not replaced by this new unit.

3. Check that STANDARD is not treated as a reserved word in selected component names.

Implementation Guideline: Declare records and packages with the name STANDARD and attempt to reveal errors by using names such as STANDARD.INTEGER or STANDARD.NUMERIC_ERROR when these names do not refer to the predefined entities.

Gaps

1. Although 8.6 says that library units mentioned in with_clauses are implicitly declared in STANDARD, it does not state when these declarations are implicitly removed from STANDARD. We assume that such declarations exist only while a library unit or one of its subunits is being compiled. Each compilation unit starts with a fresh definition of STANDARD. When compiling a library unit, the names mentioned in a with_clause are implicitly declared in STANDARD when they first occur (see 10.1.1). When compiling a subunit, the names mentioned in the with_clauses of every parent unit plus the names mentioned for the subunit being compiled are implicitly entered in STANDARD (but no such name is entered twice; see 10.1.1).

This interpretation seems to be the intended one, since otherwise, one could access any previously compiled library unit, M, as STANDARD.M even if M was not explicitly mentioned in any with clause associated with the unit being compiled.

2. Appendix C mentions some predefined library units and implies that STANDARD is also a predefined library unit. In general, it is not an error to recompile a previously compiled library unit; the newly compiled unit replaces the previously compiled unit. But because of the special nature of STANDARD (and our assumption that library units are implicitly declared in STANDARD; see Gap 3, below), it does not make sense to assume that a compiling a package called STANDARD replaces the predefined STANDARD library unit. In fact, it may be intended that no predefined library unit can be recompiled, including STANDARD (see 10.1.G). Or it may be intended only that no library unit can be named STANDARD. Or it may be intended that STANDARD is not a library unit, and hence compiling a user defined library unit named STANDARD does not imply replacement of the predefined STANDARD package. We have assumed the package STANDARD is not

a library unit and hence, that user-defined library units named STANDARD are permitted but do not replace the predefined STANDARD package.

3. 8.6 does not say that a library unit being compiled is considered to be implicitly declared in STANDARD. But the Note suggests that within a library unit, P, e.g.,

    procedure P(X: INTEGER) is ... end P;

the name STANDARD.P.X refers to the formal parameter of P, assuming the name STANDARD has not been locally redeclared. However, this statement is not supported by the sentence preceding the Note, since that sentence only mentions library units named in with_clauses. This appears to be an oversight. The sentence should have gone on to say that when a subprogram or package declaration is separately compiled as a library unit, the library unit is considered to be implicitly declared at the end of STANDARD's visible part. Similarly, when a subprogram or package body is compiled as a library unit, it is considered to be implicitly inserted in STANDARD's package body, preceded by the appropriate subprogram or package declaration. (Note: the insertion must be in STANDARD's body, since subprogram and package bodies are not permitted in a package specification.)

The proposed revised rule will explain why the identifiers declared in STANDARD are directly visibile in every library unit - it is a consequence of the normal visibility rules. But more important, as long as STANDARD is not declared locally, every identifier in a compilation unit will indeed have a unique selected component name starting with STANDARD.

4. When compiling a subunit, the LRM should say that the ancestor library unit (not subunit) is implicitly declared in STANDARD as well as all the units mentioned in any with_clauses of all ancestors of the subunit being compiled. Otherwise, these units cannot be accessed using a name beginning with STANDARD. The subunit name itself is not, of course, implicitly declared in STANDARD.

CHAPTER 9

Tasks

9.1 Task Specifications and Task Bodies

Semantic Ramifications

Neither a task declaration nor a task body can be a compilation unit (10.1). Hence, a task cannot be the main program (10.1).

An exit statement cannot cause control to leave a task body or an accept statement.

The only representation_specification forms available for tasks and task types are address_specification (13.5) and length_specification for the attribute STORAGE_SIZE (3.2). An address_specification for a task type (or the entries of a task type) is not allowed, but an address_specification (13.5) may be used to associate an entry of a task object with an interrupt.

Only the following predefined pragmas may appear in a task specification: INCLUDE, LIST, and PRIORITY.

The statement that a task specification and the corresponding body must appear in the same declarative part must be augmented by the statement (see 7.3) that if a task is declared in a package specification, its body must be declared in the corresponding package body.

Compile-time Constraints

1. A task_body must be provided for every task or task type declared in a task_specification.

2. A task_body cannot be given unless the corresponding task_specification has been given previously in the same declarative_part, or as a declarative_item in a corresponding package specification if the task_body is given in the declarative_part of a package_body.

3. Representation specifications given in a task or task type specification can only be given for the task or task type itself or for the entries declared in the task specification.

4. The identifier appearing at the end of a task declaration or body must be the same as the identifier serving as the name of the task (see Gaps).

Exceptions

An exception raised during elaboration of a task_declaration is propagated to the unit enclosing the declaration

Exceptions raised during activation or execution of the task are discussed below in Sections 9.2 through 9.11.

Test Objectives and Design Guidelines

1. Check that

   - a parameter list may not follow the identifier in a task specification.
   - a procedure type specification of the form

     <u>procedure</u> <u>type</u> identifier

     is not allowed.
   - a task cannot be a compilation unit.
   - a length specification for a task type is not allowed.
   - the identifier at the end of a task declaration or body cannot be different from the identifier serving as the name of the task (see Gaps).

2. Check that only entry declarations, the appropriate pragmas, and the appropriate representation specifications are allowed in a task specification.

   Implementation Guideline: Try an object declaration and a procedure declaration.

Gaps

1. Presumably, the identifier following the reserved word <u>end</u> must be (if present) that of the task. This is not stated in the LRM.

9.2 Task Objects and Task Types

Semantic Ramifications

Tasks are objects. They may be components of other objects and may be objects (or components of objects) designated by access types. They may be actual parameters in procedure and entry calls. Values of task types may be returned by functions (for use in naming entries or to pass as a parameter to another subprogram).

A task type may appear as the definition of a limited private type given in a private part (7.4.1). Note that in that case, the task entries (if any) are not visible outside the package.

Note that entries are not objects and hence cannot be assigned. They can, however, be renamed as procedures (see 8.5).

Compile-time Constraints

1. A task type must not be given as the type of an in out or out formal parameter, either for a subprogram nor in a generic part.

2. Task objects must not be compared for equality.

3. Task objects behave as constants and hence, cannot be used as variables in assignment statements.

4. Equality cannot be overloaded for task types nor for a composite type containing a component of a task type (see 6.7).

Test Objectives and Design Guidelines

1. Check that assignment to a record containing a task is not permissible.

2. Check that assignment to a component (for which assignment is available) of a record containing a task is available.

3. Check that a task can be passed as an actual in parameter in a subprogram call.

4. Check that an out or in out mode may not be associated with a formal of a task type (see 6.1.T/1).

5. Check that task objects cannot be assigned or compared for equality (see 5.2.1.T and 4.5.2.T), and that the equality operator cannot be defined for task types or for composite objects containing a component of a task type (see 6.7.T/1).

Gaps

1. It is assumed that for derived task types, the entry definitions and representation specifications are the same as for the parent type.

9.3 Task Execution

Semantic Ramifications

All objects (or components of objects) of a task type are activated, in an unspecified order, immediately before the first statement following the declarative part containing the object d claration (with a null statement assumed in case of a package without a body). Note however the the presence of an allocctor in the initial value of an access object that designates a task will cause task activation to take place during the elaboration of a declarative part.

If an exception arises during the (implicit) activation of the declared objects, activation will be abandoned, an unspecified subset of the objects will be activated, and the remainder will become terminated. If the unit

containing the declarative part has a handler for the exception, then the handler will receive control, otherwise no statements of the unit will be executed. In any event, the unit will not be left until the activated tasks have become terminated.

If an exception arises during the elaboration of the declarative part, then the unit will be abandoned, and the exception will be propagated to the unit that caused the activation (in the case of a task) or that invoked (in the case of a subprogram) the unit containing the declarative part in question. Any task object already elaborated will become terminated, even though it was never activated.

Note that the execution of the allocator called for in a declaration may not be postponed until the end of the elaboration of the declarative part (3.2).

When an exception arises during elaboration of a declarative part and an allocator has caused a task to be activated during the elaboration, abnormal termination of the unit causes the task object to become inaccessible (4.8), and hence, its "space" can be reclaimed. The effect of "reclaiming" such space is unclear.

Whenever a task becomes terminated, the exception TASKING_ERROR must be raised in all tasks awaiting a rendezvous with such tasks (11.5). The TASKING_ERROR exception must not be raised in tasks expecting a rendezvous with a task that does not become activated, because the elaboration of the declarative part containing the task declaration is abandoned before the elaboration of the declaration of the task in question.

Let T be a task type, and T_ACC an object of type access T; if activation of a task by the statement

```
T_ACC := new T;
```

gives rise to an exception, then the value of T_ACC will not be undefined or null, but will designate a terminated task.

Test Objectives and Design Guidelines

1. Check that declared task objects are not activated before the end of the declarative part.

2. Check that declared task objects are activated before execution of the first statement following the declarative part.

3. Check that activation of tasks created by allocators present in a declarative part takes place during elaboration of the corresponding declaration.

   Implementation Guideline: Make the task print a message when it becomes activated. Write the next declaration so that it requires a call to a function that, as a side-effect, prints a message.

4. Check that if an exception arises during implicit activation of declared task objects, the appropriate exception handler receives control.

5. Check that if an exception arises during the elaboration of a declarative part, then any tasks that were declared become terminated, while tasks whose elaboration had not yet taken place do not become terminated. Check that the TASKING_ERROR exception is raised in tasks that were awaiting a rendezvous with those tasks that become terminated; check that the exception is not raised in tasks that are awaiting a rendezvous with tasks whose declaration had not yet been elaborated.

   Implementation Guideline: The last check may be implemented as follows. After the elaboration is abandoned, remove the condition that caused the exception, and repeat the elaboration; this time these tasks that had never been activated will be able to complete the rendezvous.

Gaps

1. The technique for testing that in case of an exception during implicit activation, tasks that had been activated are not affected has not been developed. The problem is to insure that any tasks at all will be activated.

9.4 Normal Termination of Tasks

Semantic Ramifications

Note that if a task type T is declared in a library package, it does not automatically follow that every object of type T depends on the library package.

Note that a task object designated by a value of an access type depends on the block, task body or subprogram body where the access type was declared (not on the place of declaration of any particular object of that access type).

Note that an object of a limited private type may actually be a task.

Test Objectives and Design Guidelines

1. Check that a unit with dependent tasks created by object declarations is not terminated until all dependent tasks become terminated.

2. Check that a unit with dependent tasks created by allocators does not terminate until all dependent tasks are terminated.

   Implementation Guideline: To insure that the set of dependent task is not determined statically, create a linked list of records containing tasks.

3. Check that a unit terminates properly if it declares an access type designating task objects but never actually creates a task.

4. Check that a main program terminates without waiting for tasks that depend on a library package.

   Implementation Guideline: Make the library package task(s) non-terminating.

5. Check that if a task type is declared in a library package, a main program that declares objects of that type does wait for termination of such objects.

6. In a structure of the following form

```
task type T;
type T_ACC is access T;
OBJ: T_ACC;
procedure P is
begin
    OBJ := new T;
end;
```

   check that termination of the construct that activates the task (the procedure P) does not await termination of the task.

Gaps

1. A task object designated by an object of an access type may become unreachable before it terminates, e.g.

```
task type T is ... end T;
task body T is ... end T;
type T_ACC is access T;
begin
    ...
    declare
        MORIBUND: T_ACC := new T;
    begin
        null;
    end;
    -- now we've done it!
end;
```

   Termination of the inner block does not await termination of the newly created task object. According to the rules of Section 4.8, the object ceases to exist. It is not clear whether this means that the task is aborted according to the scheme in Section 9.10.

2. If a function has dependent tasks, it is not stated whether the expression in the return statement is evaluated before or after all dependent tasks are terminated.

9.5 Entries and Accept Statements

Semantic Ramifications

We assume that an entry may be called by the main program as well as from another task (see Gaps).

The language does not require that at least an accept statement be associated with each entry.

Interrupts are treated as entry calls (13.5); an interrupt entry is not executed (in response to the corresponding interrupt) unless a corresponding accept statement is executed.

The sense in which the formal_part of the accept statement must match that given in the corresponding entry declaration is defined in 6.3. A gap exists concerning the value of expressions that may appear in constraints or as default values. It is assumed here that such expressions are evaluated only once, at the time of elaboration of the entry declaration (see Gaps).

Note that an out-of-range index in the entry name does not, even if determined at compile time, make a program illegal (10.6).

An entry name can be renamed as a procedure (8.5).

An exception raised inside an accept statement and not handled locally is propagated both to the unit containing the accept statement and to the calling task (i.e., that whose call was accepted) (11.5). If the called task is terminated abnormally during a rendezvous (for example, as a result of an abort statement), the exception TASKING_ERROR is raised in the calling task at the place of call (11.5).

Termination of a task issuing an entry call does not raise an exception in the called task (11.5).

Termination of a task issuing an entry call will cause the rendezvous to be cancelled if it has not yet started (11.5).

Raising the FAILURE exception in a task that has issued an entry call does not affect the called task (11.6). If the rendezvous has not yet started, it is cancelled; if the rendezvous is in progress it is allowed to terminate.

If the FAILURE exception is received within an accept statement and not handled locally, the rendezvous is terminated and the exception TASKING_ERROR is raised in the calling task T at the place of call (11.6).

Compile-time Constraints

1. The entry_name of an accept statement must be the name of an entry family followed by an expression in parentheses, or the name of a single entry (i.e., one declared without a discrete_range).

2. A task object may accept only its own entries. This is assumed to be a compile-time constraint (see Gaps) and to require that, if the entry or entry family name in an accept statement has the form of a selected or indexed component, the prefixed name must be static. For example, the following is disallowed:

```
task type T is
    entry E (1..10);
    end;
type T_ACC is access T;
I: INTEGER := 5;
task body T is
    ...
    accept T_ACC.all.E(I) ...    -- T_ACC.all not static
    ...
    end;
```

3. Accept statements must not appear outside of task bodies.

4. Entry declarations must not appear outside of task specifications.

5. The identifier at the end of an accept statement, if present, must be that used in the declaration of the entry or entry family referenced after the word accept (see Gaps).

6. If a task's entry is renamed as a procedure inside the task body, the procedure name may be used in calls but not in accept statements (see Gaps).

7. All entries declared for a given task must have distinct names.

Exceptions

1. The exception CONSTRAINT_ERROR is raised by the evaluation of the name of an entry of a family if the index is not within the specified discrete range. (This check is suppressed by INDEX_CHECK applied to the entry family name or to the type of the index. Note: since an entry family name is neither an object nor a type, the current definition of SUPPRESS does not permit this indexing check to be suppressed. We have assumed this is an oversight.)

2. An entry call will raise the exception TASKING_ERROR (at the place of call) if the called task T terminates before accepting the call or is already terminated. (This check cannot be suppressed.)

Test Objectives and Design Guidelines

1. Check that the name of an entry family must be specified as a singly indexed component in an entry call.

2. Check that the name of a single entry (i.e. one that is not part of a family) may not be given as an indexed component in an accept statement or in an entry call.

3. Check that if the entry_name in an accept statement is a selected component, the prefix must be static (see Gaps) and must designate the task body in which the accept statement appears.

4. Check that an accept statement may not appear outside of a task body.

5. Check that the identifier at the end of an <u>accept</u> statement may be omitted.

6. Check that the identifier at the end of an accept statement, if present, must be that used in the declaration of the corresponding entry or entry family (see Gaps).

   <u>Implementation</u> <u>Guideline</u>: As an illegal case, try an accept statement for a member of a family, with the full indexed component form after the word <u>end</u>.

7. Check that if a task's entry is renamed as a procedure inside the corresponding task body, the procedure name must not be used in an <u>accept</u> statement (see Gaps).

8. Check that the exception CONSTRAINT_ERROR is raised for an out-of-range index value when referencing an entry family, either in an accept statement or an entry call.

9. Check that a task object can call its own entries and only its own entries.

10. Check that a task may contain more than one accept statement for an entry.

11. Check that a task may contain no accept statement for an entry.

12. Check that a call to an entry of a task that has not been activated does not raise exceptions.

13. Check that a rendezvous in which the accept statement has no <u>do</u> part is carried out.

<u>Gaps</u>

1. A literal interpretation of Section 9.5 would lead to the conclusion that a main program may not call entries. This seems likely to be an oversight. Note that if this rule were to be enforced, it could not be determined whether an entry call is legal in a library subprogram without knowing whether the subprogram is used as a main program.

2. It is not clear when expressions appearing in the formal part of an accept statement will be evaluated. We assume that such a formal part is not elaborated.

3. We have assumed that the identifier that follows the word <u>end</u> in an accept statement must be that used in the declaration of the entry being

accepted, or, in the case of an entry of a family, the identifier used in the family declaration.

4. The rule that "an accept statement for an entry of a given task may only appear within the sequence of statements of the corresponding task body" is not sufficient to guarantee that a task may accept only its own entries, since many task instances may share the same body. We assume that the intention was that inside a task body corresponding to a task type declaration, an accept statement may only mention the entries of "this" task object, either, by using no prefix before the entry name, or by using the type name (not an object name!). We have further assumed that the rule is intended to be checked statically.

5. The effect of renaming a task's entries inside the task body is unclear. We have assumed that such renaming will affect calls (e.g., when a task of a given type calls an entry of another task of the same type) but not accept statements (otherwise, the effect of renaming parameters and altering default values would be unclear).

9.6 Delay Statements, Duration and Time

Semantic Ramifications

Note that no upper limit is imposed on the actual duration of the delay. However, the rule in Section 9.8 on priority guarantees that at least the task with the highest priority will regain control (eventually).

Note that no guaranteed connection need exist between the elapsed time on which the delay is based and the value of TIME returned by the function CLOCK (see Gaps).

Note that the semantics of a delay statement are actually context dependent (see 9.7). This section is concerned with delay statements that are not part of a selective wait (9.7.1) or of a timed entry call (9.7.2)

If the FAILURE exception is raised by some other task, the wait is cancelled (11.6).

We assume the delay statement can appear as a statement in any context, including a main program unit (see Gaps).

Compile-time Constraints

1. The simple expression must be of the predefined type DURATION, which is a fixed point type capable of representing values between -86,400 seconds and +86,400 seconds.

Exceptions

The evaluation of the expression may give rise to exceptions. The possible exceptions depend on the form of the expression.

The FALURE exception may be raised by another task.

Test Objectives and Design Guidelines

1. Check that a negative delay is accepted.

2. Check that the minimum and maximum delays are accepted.

3. Exercise the CALENDAR package.

   Implementation Guideline: Try additions and subtractions that cause carry and borrow.

Gaps

1. For the reasons explained in Section 9.5, we have assumed that a main program may execute a delay statement.

2. It is not clear how to test that a delay statement with negative or zero value has no effect (in particular, no unnecessary context switching should take place).

3. It is not clear that the TIME returned by the CLOCK function is a reliable measure of elapsed time. In many real-time systems there is a need for both a "stamp" clock (subject to midnight "rollover" and to man-machine control) and an absolute measure of elapsed time.

4. It would be desirable to check the correct functioning of a delay that would expire past the next midnight. The practical implications of such a test are unclear.

9.7 Select Statements

9.7.1 Selective Wait Statement

Semantic Ramifications

Since a selective wait must contain at least one accept statement, and accept statements may only appear in task bodies (9.5), it follows that a selective wait may only appear in a task body.

Note that a delay statement used as a select alternative has different semantics than a delay statement in a different context (9.6).

Note that terminate is not a statement.

If the FAILURE exception is raised for a task suspended by a select statement, execution of the task is scheduled in order to allow the exception to be handled.

Compile-time Constraints

1. A selective wait must not appear outside a task body.

2. At least one select alternative must start with an accept statement.

3. At most one terminate alternative is allowed.

4. A selective wait must not contain both a terminate alternative and an alternative starting with a delay statement.

5. An else part is not permitted when either a terminate alternative or an alternative starting with a delay statement is present.

6. A terminate alternative must not appear in an inner block that declares task objects.

Exceptions

1. The exception SELECT_ERROR is raised if all alternatives are closed and there is no else part. (This check cannot be suppressed.)

Test Objectives and Design Guidelines

1. Check that a selective wait may not appear outside of a task body.

2. Check that at least one select alternative is required.

3. Check that the word select is required after end.

   Implementation Guideline: Try both

```
        end;
    and
        <<identifier>>
        select
            ...
        end identifier;
```

4. Check that terminate may not be used outside a selective wait.

5. Check that at least one select alternative must start with an accept statement.

6. Check that more than one select alternative may start with a delay statement.

7. Check that only one terminate alternative is allowed.

8. Check that only one else part is allowed.

9. Check that a terminate alternative and an else part are not allowed in the same selective wait.

10. Check that a terminate alternative and an alternative starting with a delay statement are not allowed in the same selective wait.

11. Check that an else part and an alternative starting with a delay statement are not allowed in the same selective wait.

12. Check that a delay statement is allowed in the sequence of statements of a select alternative of a selective wait containing a terminate alternative.

13. Check that all conditions are evaluated.

    Implementation Guideline: Have a select alternative without a condition, starting with an accept statement. Insure that the corresponding entry has been called, for example as follows.

```
while E'COUNT = 0 loop null; end loop;
select
    accept E;
    or when F( ) => ...
    ...
end select;
```

    Use a side effect in F to check that the condition is evaluated.

14. Check that for an open alternative starting with delay statement, the delay expression is evaluated immediately after the condition.

    Implementation Guideline: Use functions with side effects in the conditions and in the delay expressions. Use more than one (open) alternative starting with a delay statement, to guard against the case in which the condition corresponding to the alternative starting with the delay statement is evaluated last (in that case the test would not yield a meaningful result).

15. Similarly, check that the index of an accept statement for an entry family is evaluated immediately after the corresponding condition.

16. Check that the conditions are not re-evaluated during the wait.

Gaps

1. Techniques for testing the wait and the terminate alternative have not been developed.

9.7.2 Conditional Entry Call

Test Objectives and Design Guidelines

1. Check that the index and formal parameters are evaluated before the rendezvous is attempted.

Implementation Guideline: Create a task without a corresponding accept statement (then rendezvous will never be possible). Use functions with side effects to verify that the expressions are evaluated.

2. Check that if the rendezvous is not possible, the else part is executed and the sequence of statements following the entry call is not executed.

Gaps

1. The problem of testing that the rendezvous is performed if and only if it is immediately possible and has not been addressed.

2. It is not clear whether the TASKING_ERROR exception will be raised in the case that the called task is already terminated at the time of the conditional entry.

9.7.3 Timed Entry Calls

Semantic Ramifications

The semantics of the delay statement in the context of a timed entry call are similar to those of a delay statement that appears as the first statement of a select alternative in a selective wait.

If the FAILURE exception is raised during the wait, the wait is cancelled.

Test Objectives and Design Guidelines

1. Check that expressions in the actual parameters are evaluated before the rendezvous is attempted (see 9.7.2).

Gaps

1. The problem of testing that the rendezvous is not attempted if it cannot be initiated within the specified delay has not been addressed.

2. It is not clear whether the TASKING_ERROR exception is raised if the called task terminates before or during the wait.

9.8 Priorities

Semantic Ramifications

The rule in Section 9.8 of the LRM leaves some freedom in the necessary degree of pre-emption, by the use of the word "sensibly". Note, however, that whatever interpretation is taken must be consistent with the rule in Section 13.5.1 that interrupts are treated like entry calls.

Note that the expression must be evaluated at compile time (4.9); a

constraint error must raise the CONSTRAINT_ERROR exception and must not be treated as a compilation error (10.6).

Note that PRIORITY is a subtype of INTEGER (Appendix C), and hence is not overloaded on all integer types.

9.9 Task and Entry Attributes

Semantic Ramifications

PRIORITY is a subtype of INTEGER (Appendix C).

Note that the priority of a task is static (9.8); tasks of a task type all have the same priority. Therefore the task name T in T'PRIORITY need not be static, since its type is always known.

The exception CONSTRAINT_ERROR is raised if task is not allocated (4.1.3).

Compile-time Constraints

1. TERMINATED is not applicable to task types (see Gaps).

2. COUNT may be used only for an entry of a task T inside the body of T.

Test Objectives and Design Guidelines

1. Check that TERMINATED may not be used for a task type.

2. Check that a task may not interrogate the COUNT of another task's entry.

3. Check that for an entry of a family the entry name must have the form of an indexed component.

4. Check that the task name need not be static.

Gaps

1. We have assumed that TERMINATED is not applicable to a task type.

9.10 Abort Statement

Test Objectives and Design Guidelines

1. Check that at least one task name is required.

2. Check that the task name need not be static.

3. Check that aborting a terminated task does not cause exceptions.

4. Check that if a task is aborted before being activated, the task is terminated.

5. Check that after the abort statement, the named tasks and all dependent tasks are terminated.

6. Check that a task may abort itself; check that the next statement is not executed.

   Implementation Guideline: The abort must be conditional, otherwise the compiler may delete all following statements.

7. Check that a task may abort a task that it depends on.

8. Check that an aborted task is removed from any entry queue it may be on.

CHAPTER 10

Program Structure and Compilation Issues

10.1 Compilation Units -- Library Units

Semantic Ramifications

Note that a compilation unit can begin with more than one with_clause. A compiler therefore could provide an optional compilation mode in which access to certain library units was implicitly provided by (in effect) inserting a with_clause (and optionally, a use_clause) naming these units in front of every unit compiled. These implicit insertions would be intended to provide automatic access to a standard set of library units, e.g., the TEXT_IO package, a package of mathematical functions, or an application-dependent set of library routines. Such a capability would have the same effect as invoking a preprocessor that modifies compilation units before they are compiled. Nothing in the language specification forbids or requires such a "preprocessor" option, but it is one of the reasons more than one with_clause is permitted at the beginning of compilation units. Of course, if such an option is provided, there must be a way to turn it off or the implementation will be non-standard.

Note that although the syntax says a compilation may consist of zero compilation_units, the text says one or more units are required.

The requirement that library units have unique names implies that subprogram library units cannot be overloaded; recompilation of a subprogram with a different set of parameters simply replaces the earlier definition (including its subunits, if any; see 10.3). Note also that since recompilation redefines the associated library unit, a package can be recompiled as a subprogram, and vice versa.

Certain library units are predefined in the language (see Appendix C) -- CALENDAR, SHARED_VARIABLE_UPDATE, UNCHECKED_DEALLOCATION, UNCHECKED_CONVERSION, INPUT_OUTPUT, TEXT_IO, and LOW_LEVEL_IO. Since there is no rule forbidding recompilation of these units, it is presumably permitted, even though such a recompilation would make the predefined units inaccessible.

If library units as well as units mentioned in with_clauses are implicitly declared in STANDARD when they are compiled (see 8.6.G), no library unit can have the same name as any identifier declared in STANDARD (since this would introduce a forbidden redeclaration in STANDARD). Moreover, if a library unit is implicitly declared in STANDARD while being compiled, then a with_clause cannot also contain the name of the (library) unit being compiled, e.g.,

```
with P;
package P is    -- illegal redeclaration in STANDARD.
```

Similarly,

```
with P;
package body P is        -- illegal redeclaration in STANDARD.
```

Finally, since the most remote ancestor of a subunit must be considered to be implicitly declared in STANDARD when a subunit is compiled (see 8.6.G/4), the following is also illegal:

```
with P;
separate (P.Q.R)         -- implies illegal redeclaration in STANDARD
package body S is
```

Note that the names in a use_clause do not have to have been mentioned in the immediately preceding with_clause; any package name in any preceding with_clause can be mentioned. Moreover, since the with_clauses given for a compilation unit apply it to its subunits (and since subunits are also compilation units), to subunits of subunits, etc., a use_clause can mention the name of a package appearing in any ancestor's with_clause, as long as such a name is not hidden at the point where the use_clause appears. Finally, note that the rule specifying what package names can appear in a context_specification's use_clause excludes package names whose visibility is achieved by direct declaration instead of by the effect of preceding with_clauses, use_clauses, or declarations in STANDARD. The following examples illustrate these points.

```
package B is
    type TB is ...
end B;
----------
package A is
    package PA is
        type TA is ...
    end PA;
    package RR is ... end RR;
end A;
----------
with A;                  -- (1)
package P is
    X: A.PA.TA;
end P;
----------
with B; use A;           -- (2); the with at (1) applies to body
package body P is
    package QQ is        -- effect at (3) is the same if use A is given
                         --    inside P before QQ's declaration
        type TQ is ... ;
    end QQ;
    package RR is ... end RR;    -- hides A.RR
    procedure PP(X: PA.TA; Y: B.TB; Z: QQ.TQ) is separate;
end P;
----------
```

```
with A; use B, PA, RR;  -- (3)
separate (P)
procedure PP (X: TA; Y: TB; Z: QQ.TQ) is
    use QQ;       -- P.QQ
    ZZ: TQ
```

The with_clause at (3) is required, even though it adds no information, because a use_clause cannot be written as the sole constituent of a context specification. Note that at position (3), you can write PA instead of A.PA, since the use_clause associated with package body P also applies to this subunit. In addition, note that even though QQ is directly visible at (3), you cannot write <u>use</u> B, PA, QQ, since the LRM says that the only package names permitted in a use_clause of a context_specification are those declared in STANDARD and those made visible by previous with_clauses and use_clauses. QQ (and RR) are directly visible by virtue of their declaration in package body P, and hence do not satisfy any of the criteria permitting them to be mentioned in (3)'s use_clause. Hence, RR in (3) refers to A.RR, not P.RR. Package A.PA in A has been made directly visible by a preceding use clause, and so can be mentioned in (3). And at (2), we could not write <u>with</u> B; <u>use</u> A, A.PA, since A.PA has not been made directly visible by a use_clause. However, at (3) we can write A.PA or just PA, since the use_clause at (2) makes PA directly visible at (3).

We suspect (see Gaps) that the intent of the rule regarding use_clauses in context_specifications was merely to state the consequences of the usual visibility rules and not to impose an additional restriction that forbids the use of QQ in (3), forbids A.PA at (3), and implies that RR at (3) refers to A.RR.

<u>Compile-time Constraints</u>

1. A subprogram that is a library unit or subunit cannot have an operator_symbol as its designator.

2. No library unit can have a name identical to an identifier declared in STANDARD (see T/1b for a list of these names).

3. A package body must not be compiled before its specification has been compiled.

4. A directly visible package cannot be named in a context_specification's use_clause unless it is a package declared in STANDARD or has been made directly visible as the result of a previous with_clause or use_clause (see Gaps).

5. Only a library unit subprogram can executed as a main program.

6. The name of the unit being compiled cannot appear in a with_clause associated with that unit (see Gaps).

7. The name of the most remote ancestor of a subunit cannot appear in a with_clause for that subunit (see Gaps).

Test Objectives and Design Guidelines

i. Check that

a. a subprogram cannot be compiled as a library unit or subunit if its designator is an operator_symbol.

b. no library unit can be named BOOLEAN, INTEGER, ABS, FLOAT, CHARACTER, ASCII, NATURAL, PRIORITY, STRING, DURATION, CONSTRAINT_ERROR, NUMERIC_ERROR, SELECT_ERROR, STORAGE_ERROR, TASKING_ERROR, or SYSTEM (see Gaps).

c. a package body cannot be compiled before its specification has been compiled.

d. consecutive use_clauses are not permitted in a context_specification.

e. a use_clause cannot name a subprogram mentioned in a preceding with_clause (see LRM 8.4).

f. a use_clause cannot appear by itself as a context_specification.

g. a use_clause cannot mention a package name declared in STANDARD (namely, ASCII and SYSTEM) if these names have been hidden by some more local declaration.

h. a use_clause in a context_specification cannot mention a directly visible package unless it is a package declared in STANDARD or has been made directly visible as the result of a previous with_clause or use_clause.

<u>Implementation Guideline</u>: Try using names of previously compiled units that have not been mentioned in previous <u>use</u> or <u>with</u> clauses. Also try using a name of a package nested inside a package mentioned in a with_clause. Also try using (in a subunit) a package name that is directly visible but not in STANDARD (see Gaps). Finally, try <u>with</u> B; <u>use</u> A; <u>with</u> A; where A has not been previously mentioned in a with_clause.

i. a with_clause must be terminated by a semicolon, e.g.,

<u>with</u> A <u>use</u> A; -- illegal

j. the names in a with_clause cannot be names of uncompiled library units, nor can they be names of visible subunits or nested packages (see 10.1.1.T/1).

k. an implicit with_clause and use_clause is not provided for any of the standard packages, namely, CALENDAR, INPUT_OUTPUT, TEXT_IO, and LOW_LEVEL_IO (see LRM Appendix C).

l. an implicit with_clause is not provided for any of the predefined

units CALENDAR, SHARED_VARIABLE_UPDATE, UNCHECKED_DEALLOCATION, UNCHECKED_CONVERSION, INPUT_OUTPUT, TEXT_IO, and LOW_LEVEL_IO (see LRM Appendix C).

m. the name of the unit being compiled (or the most remote ancestor of a subunit) cannot be the same as a name mentioned in a with_clause (see Gaps) (see 10.1.1.T/1e).

n. overloaded subprograms are not entered into the library as distinct units.

o. an implementation does not provide implicit access (via with and use clauses) to user-defined library units.

p. task specifications and bodies cannot be separately compiled.

q. a subunit that is a subprogram cannot be initiated as a main program.

2. Check that a subunit can have the same name as a library unit or an identifier declared in STANDARD.

3. Check that more than one completely independent compilation unit can be submitted to a compiler in a single file and that the presence of an illegal unit in a compilation does not affect the legality of another, independent unit submitted in the same compilation.

4. Check that a package specification and body can be submitted together for compilation.

5. Check that a subprogram specification and body can be submitted together for compilation.

6. Check that a library unit and its subunits can be submitted together for compilation.

7. Check that a non-generic package specification can be compiled without its body (implicitly checked by other tests).

8. Check that a generic package specification can be compiled without its body (see 12.1.T/?).

9. Check that a non-generic subprogram declaration can be compiled separately.

10. Check that a generic subprogram declaration can be compiled separately (see 12.1.T/?).

11. Check that a non-generic subprogram body can be submitted for compilation without having previously submitted a subprogam declaration or body (checked implicitly by other tests.

12. Check that a generic subprogram body can be submitted separately from its declaration for compilation (see 12.1.T/?).

13. Check that a generic package or subprogram instantiation can be submitted for compilation (see 12.3.T/?).

14. Check that a subunit can be submitted for compilation separately from its parent unit.

15. Check that a use clause in a context_specification can name the ASCII and SYSTEM packages declared in STANDARD.

16. Check that more than one with_clause can appear in a context specification.

    Check that use_clauses can mention names made visible by preceding with_clauses in the same context_specification.

17. Check that a use_clause in a context_specification associated with a package body or subprogram body can mention the name of a package, e.g., P, named in its separately compiled specification, even if P is not named in the context specification for the body being compiled and is not otherwise directly visible.

18. Check that a use_clause in a context_specification associated with a subunit can mention any package named in any of its ancestor's with_clauses.

    Implementation Guideline: Attempt to name packages made visible by the immediate ancestor of the subunit as well as more remote ancestors, and also, if the remotest ancestor's declaration was separately compiled, attempt to use a package name specified explicitly with the declaration but not with the corresponding body.

19. Check that in a compilation containing several compilation units, the order of elaboration need not be the same as the order of compilation (see 10.5.T/2).

20. Check that a subprogram can be recompiled as a package (and vice versa).

    Check that if the previously compiled unit had subunits, these subunits are no longer accessible.

21. Check that any of the predefined library units can be recompiled, making the redefined units inaccessible.

Gaps

1. The LRM forbids naming directly visible packages in a context_specification's use_clause unless they are directly visible because of their declaration in STANDARD or by the effect of previous with_clauses and use_clauses. Direct visibility because of explicit declaration in an ancestor unit or by selected component naming starting from a package named in a previous with_clause (or use_clause) is not permitted. This appears to be an oversight, since it complicates the

visibility rules for use_clauses in context_specifications. However, the statement in LRM 10.1 is very clear, so our tests are constructed to check that compilers implement the stated restriction. Nonetheless, it seems likely to us that this restriction will be removed before the current version of the Standard is finalized.

2. The restrictions (C/6, 7) that the name of a unit being compiled or the name of the most remote ancestor of a subunit cannot appear in a with_clause stems from our opinion that the names of these units are to be considered implicitly declared in STANDARD (see 8.6.G). Similarly, our constraint that the names of units cannot be identical to identifiers declared in STANDARD stems from this same opinion.

3. The first and second paragraphs on page 10-2 contain the phrases "its context is" and "a context are". In both cases, the word "context" should be replaced with "context specification". (In an earlier draft of the LRM, the term "context_specification" was just "context". Not all references to the earlier terminology seem to have been changed appropriately.)

10.1.1 With Clauses

Note that a unit name (not identifier) can be used in a with_clause. However, unless a package or subprogram has been previously mentioned in a with_clause, the only form of name that exists is "identifier". After a unit has been mentioned in a with_clause, however, it is implicitly declared in STANDARD and so can be named as STANDARD.identifier. This appears to be the only form of name other than identifier that can appear in a with_clause.

Because the names mentioned in with_clauses will not necessarily be declared implicitly in STANDARD in the order they appear in the with_clause, a programmer cannot specify the order of elaboration of units by the order in which they are named in with_clauses (see 10.5.S). Nor does an implementer have to check that the with_clause names appear in an order satisfying the compilation order rules (see 10.3). In short, the effect of the statement regarding order of declaration in STANDARD is to remove the need for checks that would otherwise have to be made.

Note that although the specification says that duplicate names are permitted in different with_clauses of a single context_specification, it does not permit duplicate names within a given with_clause (since such a duplication would imply duplicate declarations in STANDARD). The specification similarly does not permit a with_clause in a subunit's context specification to mention any unit named in the context_specification of any ancestor. However, we consider this an oversight (see G/1).

Note that since a with_clause has the effect of declaring library units in STANDARD, it does not hide identically named entities that are otherwise directly visible, e.g.;

```
package P is
    package R is
        X: INTEGER := 5;
    end R;
end P;

with R; -- library unit R, not P.R
package body P is
    Y: INTEGER := R.X;  -- P.R.X, not STANDARD.R.X
```

Compile-time Constraints

1. The names appearing in with_clauses can only be names of (predefined or previously compiled) library units.

2. A unit cannot be named more than once in a given with_clause.

Test Objectives and Design Guidelines

1. Check that

   a. The names in a with_clause cannot be names of previously uncompiled library units or names of packages appearing in STANDARD.

   b. a visible (package) subunit or nested package cannot be named in a with_clause.

   c. a library unit, P, cannot be named in a with_clause as STANDARD.P (if it has not been named in a previous with_clause associated with the unit being compiled).

   d. with STANDARD; is not permitted if there is no user-defined library unit called STANDARD.

   e. the name of the unit being compiled (including the most remote ancestor of a subunit) cannot be the same as a name in a with_clause.

   f. a with_clause cannot mention a unit name more than once.

2. Check that any of the predefined library unit names, CALENDAR, SHARED_VARIABLE_UPDATE, UNCHECKED_DEALLOCATION, UNCHECKED_CONVERSION, INPUT_OUTPUT, TEXT_IO, and LOW_LEVEL_IO, can be specified in a with_clause for the first unit compiled in a new library (and check that access is provided to the entities declared in these packages).

3. Check that a with_clause provides access to previously compiled non-generic subprogram and package library units.

4. Check that a generic package or subprogram named in a with_clause can be instantiated (see 12.3.T/?).

5. Check that a with_clause can name a library unit specified in a previous with_clause appearing in the same context_specification.

Check that a with_clause associated with a subunit can name a library unit specified in a with_clause of any ancestor unit (see Gaps).

6. Check that if C is a unit named in a previous with_clause, the unit can be named as STANDARD.C in subsequent with_clauses.

   Check that within the unit being compiled, any unit named in a with_clause can be referenced as a unit declared in STANDARD, including units named in with_clauses of a subunit's ancestors.

7. Check that if a with_clause in a subunit contains a name, e.g., RR, that has been locally declared in one of the subunit's ancestors, the with_clause's name is nonetheless considered to refer to a library unit, and the library unit can be accessed as STANDARD.RR.

Gaps

1. We have assumed that if a with_clause associated with a subunit contains a library unit name mentioned previously in a context_specification associated with an ancestor unit, the repeated name is not considered illegal, but is ignored as though all names in all the applicable context specifications had appeared in a single context specification. The effect of with_clauses associated with subunits is not explicitly addressed in 10.1.1. LRM 10.2 says that "The context (specification) of the subunit may mention additional library units." It does not state that the context specification may only mention additional units. Although the phrasing could be taken to indicate that "may only" was intended, we have assumed that duplication would be harmless. In any event, the LRM does not clearly resolve this point.

2. At the end of the first paragraph, the phrase "a given context" appears. It should read "a given context specification".

10.2 Subunits of Compilation Units

Semantic Ramifications

Note that since subunit names of a given parent must be distinct, body_stubs cannot introduce two or more overloaded subprogram bodies within the same declarative part.

We assume (see Gaps) that although the subunits of a parent unit must be named uniquely, subunits of different parents having a common ancestor can have identical identifiers as their names, since the full name of the subunit will always be distinct.

Since separately compiled subprograms cannot have operator symbol names (see 10.1) no subprogram whose designator (see 6.1) is an operator_symbol can have its body specified with a body_stub.

Because subunits are compiled as though they were substituted for the

corresponding body stub, an implementation must be careful to ensure there is no visibility within a subunit of entities declared after the corresponding body_stub. The only potentially visible entities that can appear after a body stub are package and task declarations (see 3.9). Hence, one can have:

```
package P is
    Y: INTEGER := 6;
end P;

package body P is
    X: INTEGER;
    procedure Q is separate;
    package P is
        Y: INTEGER := 5;
    end P;
    procedure R is separate;
end P;

separate (P)
procedure Q is
    Z: INTEGER := P.Y;  -- not P.P.Y
    ...

separate (P)
procedure R is
    Z: INTEGER := P.Y   -- P.P.Y
```

Note that within Q, there is no visibility of package P.P, but procedure R does have visibility of P.P, so within procedure R, P.Y refers to P.P.Y.

Since the with_clause of a parent unit, P, is visible within a subunit, Q, if P is also a subunit, P has visibility of its parent's with_clause as well. Hence, Q has visibility of all the with_clauses of its ancestors. Of course, the names of units introduced by with_clauses can be hidden by more local declarations, as a consequence of normal visibility rules and as illustrated by examples in 10.1.S and 10.1.1.S.

The note (in the LRM) about renaming declarations refers to the following situation:

```
package S is
    procedure Q1 (A; INTEGER):
    procedure Q2 (B: FLOAT);
end S;

with S; use S;
package P is
    procedure Q (A: INTEGER) renames Q1;
    procedure Q (B: FLOAT) renames Q2;  -- overloading introduced
end P;
```

```
package body S is
    procedure Q1 (A: INTEGER) is separate;
    procedure Q2 (B: FLOAT) is separate;
end P;
```

Note that the following would not be permitted:

```
package P is
    procedure Q (A: INTEGER);
    procedure Q (B: FLOAT);
end P;

package body P is
    procedure Q1 (A: INTEGER) renames Q;
    procedure Q2 (B: FLOAT) renames Q;
    procedure Q1 (A: INTEGER) is separate;
    procedure Q2 (B: FLOAT) is separate;
end P;
```

This is illegal since Q1 and Q2 are declared twice in package body P.

It is not clear whether the following is forbidden:

```
package P is
    procedure Q (A: INTEGER);
    procedure Q (B: FLOAT);
private
    procedure Q1 (A: INTEGER) renames Q;
    procedure Q2 (B: FLOAT) renames Q;
end P;

package body P is
    procedure Q1 (A: INTEGER) is separate;
    procedure Q2 (B: INTEGER) is separate;
end P;
```

Does the definition of Q1 and Q2 in P's body satisfy the requirement that each Q be given a body? The answer is unclear (see 8.6.G/19).

The order of subunit elaboration is determined by the order in which body stubs appear in a parent. Therefore, any initializations caused by the subunit elaborations take place in the order specified by the body stubs. For example,

```
package body P is
    X: INTEGER;
    package body Q is separate;
    package body R is separate;
end P;
```

```
separate (P)
package body R is
begin
    X := 5;      -- P.X
end R;

separate (P)
package body Q is
begin
    X := 6;      -- P.X
end Q;
```

Even if these three units are submittted together in a single compilation, after P is elaborated, P.X has the value 5, since package body R is elaborated after package body Q. This example also illustrates the fact that subunits do not have to be compiled in the order of their body stub declarations.

Compile-time Constraints

1. A body_stub is permitted only in the outermost declarative part of a compilation unit.

2. A subunit must give the full name of the parent unit, i.e., the first identifier in the unit_name must be the name of a library unit and the last identifier must be the name of the unit containing the body stub for the unit being compiled.

3. A body_stub cannot be given for two or more overloaded subprograms in the same declarative part.

4. A body_stub cannot be given for a subprogram named with an operator_symbol.

5. Two or more body_stubs cannot be given for the same subunit.

6. A body_stub is forbidden if a corresponding subprogram, package, or task declaration has not appeared earlier in the same declarative part (see Gaps).

7. The rules for agreement between the subprogram_specification appearing in a body_stub and that appearing in the corresponding subprogram declaration are the same as those for subprogram_bodies (see 6.3.C) (see Gaps).

8. The rules for agreement between the subprogram_specification appearing in a body_stub and that appearing in the corresponding compilation unit are the same as those specified in C/7 (see Gaps).

Test Objectives and Design Guidelines

1. Check that

   a. a body_stub cannot be given a declarative part physically enclosed by

another declarative part, e.g., in a procedure nested in a package body.

b. a task, subprogram, or package body_stub cannot be given unless there has been a preceding specification for the stub (see Gaps).

c. two body stubs in the same declarative part cannot contain overloaded subprogram specifications.

d. a body_stub cannot be given for a subprogram named with an operator_symbol.

e. two body stubs cannot be given for the same compilation unit.

f. if the identifier of a subunit is unique in a program library and it is the parent of another subunit, the unit_name following <u>separate</u> cannot be just the name of the parent.

g. if P is the full name of a subunit's parent, check that STANDARD.P cannot be used as the name following <u>separate</u>.

h. if a body_stub is deleted from a compilation unit, the previously existing subunit can no longer be accessed (see also 10.1.T/20).

2. Check that subunits of different parents having a common ancestor can be named with identical identifiers (see Gaps).

3. Check that a subunit has visibility of identifiers declared prior to its body_stub but not those declared after its body stub (see also 10.1.T/18).

4. Check that a subunit has visibility of identifiers declared in ancestors other than the parent.

5. Check that a with_clause associated with a subunit can name a library unit specified in a with_clause of any ancestor unit (see 10.1.T/5 and 10.1.G/1).

6. Check that if a with_clause for a subunit contains a name that is directly visible in one of the subunit's ancestors, the with_clause name is nonetheless considered to refer to a library unit, not the locally declared entity (see 10.1.1.T/7).

7. Check that subunits are elaborated in the order in which their body stubs appear, not (necessarily) in the order in which they are compiled.

8. Check that if an overloaded subprogram is declared, one of the subprogram bodies can be specified with a body stub and compiled separately.

9. Check that a generic subunit can be specified and instantiated (see 12.3.T/?).

10. Check that subunit names can be identical to identifiers declared in STANDARD.

Gaps

1. Although the LRM requires that names of all subunit descendants from a given library unit be distinct, the LRM does not make clear whether the restriction applies just to the identifiers naming the subunits or to the full names, e.g., are all the following permitted as subunit names: P.Q, P.R, P.Q.R, and P.R.Q? Since full names must be given in the subunit specification, there seems to be no advantage in requiring that all subunit descendants of a library unit be given unique <u>identifiers</u> as names. It is sufficient if a unique full name exists for a subunit of a given parent.

2. In the second paragraph after the syntax, the phrase "The context of the subunit" appears. This should have read "The context specification of the subunit". Similarly, the last sentence of this paragraph says "the subprogram body is elaborated." It clearly should have said "the subunit body is elaborated".

3. The LRM says a subunit's context_specification can name additional library units. It does not say that <u>only</u> previously unnamed library units can be named. We have assumed, therefore, that duplicate names are ignored (see 10.1.1.C).

4. A body_stub for a subprogram is not stated to act as the declaration of the subprogram if no explicit declaration has been given previously. The implicit declaration allowed for subprogram bodies is only explicitly permitted for subprogram_bodies (6.3) and compilation_units (10.1). Forbidding body_stubs to act as declarations seems to be an oversight, but since the wording is clear, our tests check that this rule is enforced.

5. Nothing is said about how the subprogram_specification given in a body stub must agree with the specification appearing in the subprogram's declaration, or in the corresponding compilation unit. We have assumed that the same rules apply as for subprogram_bodies (see 6.3.S).

10.3 Order of Compilation

A library need not be complete or consistent (with regard to recompilation) in order to begin execution of a main program if the main program does not potentially make use of any of the missing or obsolete units (see 10.4.S).

Note that recompilation of units that are modified only with respect to their comments or formatting do not actually require recompilation of dependent units, although a simple implementation may still require their recompilation. Similarly, if a package specification is augmented with declarations at the end, no offset addresses for declared items will (in many implementations) be changed. In such cases, it is unnecessary to recompile dependent units that do not access these new declarations.

Compile-time Constraints

1. A package or subprogram declaration (generic or non-generic) cannot be compiled until the units mentioned in its with_clauses have been compiled.

2. A package body cannot be compiled until its specification has been compiled and until any units mentioned in its with_clauses have been compiled.

3. A subprogram body cannot be compiled until any units mentioned in its with_clauses have been compiled.

4. A subunit cannot be compiled until its parent has been compiled and until any units mentioned in its with_clauses have been compiled.

5. If a package or subprogram declaration is modified in a way that invalidates the correctness of code generated for units using the modified package or subprogram, then all these units must be recompiled before execution of the main program is permitted.

6. If a package body or subprogram body (generic or non-generic) is modified in a way that invalidates the correctness of code generated for units using these bodies, then all these units must be recompiled before execution of the main program is permitted.

7. If the parent of a subunit is modified in a way that affects the correctness of code generated for the subunit, then the subunit must be recompiled before execution of the main program is permitted.

Test Objectives and Design Guidelines

1. Check that

   a. a package or subprogram declaration (generic or non-generic) cannot be compiled if the units mentioned in its with_clauses have not been compiled.

   b. a package body cannot be compiled if its specification has not been compiled.

   c. a package body cannot be compiled if any units mentioned in its with_clauses have not been compiled.

   d. a subprogram body cannot be compiled if any units mentioned in its with_clauses have not been compiled.

   e. a subunit cannot be compiled if its parent has not been compiled.

   Implementation Guideline: Use a subunit having more than one ancestor as well as a subunit with a singl ancestor.

   f. a subunit cannot be compiled if any units mentioned in its with_clauses have not been compiled.

2. Check that a unit can be compiled if a package it uses has been recompiled but the corresponding package body has not been recompiled, even if the unit uses a subprogram defined in the yet-to-be recompiled body.

3. Check whether recompiling a package specification or subunit parent that is changed only with respect to its comments or formatting nonetheless requires recompilation of all subordinate units.

   Check whether recompiling a package specification to which declarations have been added at the end causes recompilation of subordinate units to be required.

4. Check that if the body of an INLINE subprogram is modified, all units calling that subprogram as an inline subprogram must be recompiled.

5. Check that if a separately compiled generic unit is substantively modified, all units instantiating it must also be recompiled.

6. Check that if a package specification is substantively modified (e.g., by changing declarations in a way that changes their type or size), previously compiled units using the modified declarations must be recompiled.

7. Check that if a subprogram declaration is substantively modified, all units invoking that subprogram must be recompiled and so must the subprogram body.

8. Check that if the parent of a subunit is substantively modified, all its subunits must be recompiled.

10.4 Program Library

Semantic Ramifications

The commands for reusing units of other program libraries permit an implementation to provide a way of adding units to a program library without submitting them first for compilation. Such a facility might be used to add application-oriented packages to a particular library of Ada programs. Or, if the program library interfaces with a version control system, these commands might be used to indicate which versions of program units are to be used when compiling some unit or when initiating execution of a main program.

The commands for interrogating the status of program library units could be used to indicate which units require recompilation, which units are missing, which units are unreferenced, etc.

Test Objectives and Design Guidelines

Since these commands are optional capabilities, no tests are provided at this time.

10.5 Elaboration of Library Units

Semantic Ramifications

Elaboration order dependences induced by subprogram calls stem from the rule (given in LRM 3.9) that a subprogram cannot be invoked until its body has been elaborated:

```
package A is
    procedure F;
end A;

package B is
    procedure G;
end B;

with B;
package body A is
    procedure F is ... end F;
begin
    B.G;        -- must elaborate B's body before A's
end A;

with A;
package body B is
    procedure G is  ... end G;
begin
    A.F;        -- must elaborate A's body before B's
end B;
```

If the with_clauses and the call to B.G in package body A are removed for the moment, all four units above could be combined in a single declarative part without violating 3.9's rule, since the invocation of A.F will occur when package body B is elaborated, which will occur after package body A is elaborated, since package body A would precede B's body in the declarative part.

When the units are separately compiled, either package body A or B can be compiled first according to the compilation order rules (10.3). (The compilation ordering rules permit either body A or B to be compiled first once both package specifications have been compiled.) But regardless of the compilation order, if the call to B.G is eliminated, package body A must be elaborated before package body B to ensure F's body is elaborated before it is called in package body B. If the call to B.G is left in package body A, then there is no consistent order of elaboration. Any attempt to execute a main program using packages A or B will be illegal.

The statements in a package body can be used to initialize variables and data structures. For example:

```
package A is
    X: INTEGER;
end A;

package body A is
begin
    X := 5;
end A;
```

Can some unit using the visible part of A depend on A.X having an initial value? For example, if we have:

```
with A;
package B is
    Y: INTEGER := A.X;
end B;
```

can a programmer be sure that B.Y has A.X's initial value? Must an implementer ensure that A's body is elaborated before B is elaborated? Since using an uninitialized variable makes a program erroneous, not illegal (see 3.2), there appears to be (see Gaps) no requirement for an implementation to ensure package bodies that initialize data are elaborated before package bodies that use the initialized data.

In short, an implementation need only keep track of which library units call subprograms declared in other packages, and it must ensure that the library units containing the called subprogram bodies are elaborated before the units containing the subprogram calls. Note that such calls can occur in the declarative items of a package specification, the declarative part or statements of a package body, and the declarative part or statements of a subprogram body. Note that if package body A calls B.G, and B.G calls C.H, then C must be elaborated before B, and B before A, i.e., an implementation must take into account dependences caused by chains of calls.

Which library units must be elaborated depends on which unit is selected to be the main program. Consider all the library units named in with_clauses of the main program, the library units named by these library units, etc. The set of library units named directly and indirectly by the with_clauses of the main program determine what library units must be elaborated prior to calling the main program, and of course, all subunits of these library units must be elaborated.

If certain library units are not required by the execution of a main program, then it does not matter whether these library units are present in a library or if they are in a consistent state with respect to recompilation rules. For example, if package specification A has been recompiled but its body has not yet been recompiled, a main program can still be called if it does not reference package A.

Similarly, if as a result of conditional compilation (see 10.6), the elaboration of certain library units is not required, an implementation is presumably free (see Gaps) to execute the main program without elaborating the

unneeded unit. In such cases, one implementation could permit execution of a main program that another implementation would consider illegal. For example, suppose in the first example of this section, we had written:

```
    if FALSE then B.G;
```

in the body of package A. If the compiler exercises its right to conditionally compile code, then there is no call to B.G in the body of package A, and hence, no circular package body references exist. An implementation that did not take advantage of conditional compilation opportunities, however, would be free to reject an attempt to elaborate A and B on the basis that no consistent order of elaboration exists.

Compile-time Constraints

1. A program is illegal if, when attempting to invoke the main program, there is no ordering of package body elaborations that permits a subprogram body to be elaborated before it is called from another library unit.

Test Objectives and Design Guidelines

1. Check that a main program cannot be invoked if its elaboration requires an inconsistent elaboration order of library units it uses due to subprogram invocations.

   Implementation Guideline: Use calling chains of length greater than one.

2. Check that when several compilation units are submitted to a compiler in a single compilation, the order of compilation need not be the same as the required order of elaboration (use calling chains of length greater than one).

   Repeat this check when the units are submitted in separate compilations.

3. Check that the elaboration of library units required by a main program is performed consistently with the partial ordering defined by the compilation order rules.

4. Check whether conditional compilation affects the legality of a particular set of library units and the order in which they can be elaborated (see Gaps).

5. Check that if two package bodies initialize the same package variable, an implementation permits either package body to be elaborated first, i.e., it does not reject as illegal a program using these packages.

6. Check whether if one package body initializes a package variable and other package bodies use the variable, the initializing package body is elaborated first.

   Implementation Guideline: To minimize accidentally elaborting the initializing package body first, it should be compiled neither first nor

last and it should not be the package body associated with the initialized package variable. Its name should also not be alphabetically first or last among the units being compiled.

Gaps

1. It would be helpful if the specification clearly stated that the only dependence relations affecting order of elaboration are those induced by subprogram calling relationships. As it is, one is left uncertain as to whether access and assignment relationships also affect elaboration order. We have assumed that assignments to package variables from a package body do not induce a required elaboration order dependence.

2. It is uncertain whether the effect of conditional compilation can be considered when determining elaboration order.

CHAPTER 11

Exceptions

11.1 Exception Declarations

Semantic Ramifications

Exception names in Ada follow Ada's normal scope and visibility rules (see Sections 8.2 and 8.3). Hence, a local exception can be declared using the same identifier as a pre-defined exception, and the local exception is distinct from the predefined exception. Note also that since predefined exceptions are all declared in the package STANDARD, STANDARD.NUMERIC_ERROR is a notation that can be used almost always (see 8.3.h and 8.6) to access the predefined declaration of NUMERIC_ERROR, even when NUMERIC_ERROR has been declared more locally as a user-defined exception. And if STANDARD is a subunit (see 10.2), it is possible for STANDARD.NUMERIC_ERROR to refer to a user-defined exception. Hence, an implementation's method for identifying an exception cannot depend solely on how the exception is spelled.

Note that if an exception is declared in the visible part of a package, it can be referenced by any module importing that package. An implementation must ensure that different modules importing this package treat the package's exception as the same exception (see 8.5).

Note that if a generic package declares an exception, each instantiation of the package introduces a new exception (since each instantiation introduces a new set of declarations). Note that since a task specification cannot contain an exception declaration, there are no user-defined exceptions associated with task types. There is only one predefined exception, FAILURE, associated with all task objects and types.

It is possible for an exception to be propagated (see 11.4.S) out of the scope of its declaration (e.g., into a subprogram that is unable to name the exception explicitly) and then have the exception propagated back into its scope. This means that in general, all exceptions must have unique run-time identifications. It is a consequence of the semantics for FAILURE (see 11.6) that FAILURE need have only one physical representation, regardless of the number of tasks in a given program.

The exception STORAGE_ERROR need not be supported by an implementation (see 3.9). Note also that STORAGE_ERROR may be raised by the main program (see 3.9).

Test Objectives and Design Guidelines

1. Check that all predefined exceptions (CONSTRAINT_ERROR, NUMERIC_ERROR, SELECT_ERROR, STORAGE_ERROR, TASKING_ERROR, and FAILURE (see 11.6.T)) may be raised explicitly with raise statements and may have handlers written for them.

2. For each predefined exception name, check that the name is not a reserved word.

   Implementation Guideline: Check not only that the name can be redeclared as a variable, but also that the redeclared name cannot still be used in a raise statement (see also 11.2.T/1 and 11.3.T/1).

3. Check that a programmer-defined exception having the same name as a predefined exception is distinct from the predefined exception.

   Implementation Guideline: In a simple hierarchy of nested blocks (each block nested inside another), the innermost block should contain a handler for NUMERIC_ERROR (or CONSTRAINT_ERROR) and a declaration redefining NUMERIC_ERROR as an exception. Two other blocks in the propagation path of the exception should also contain handlers for NUMERIC_ERROR. The innermost of these 3 blocks should contain a statement raising NUMERIC_ERROR by propagation. The second of the 3 handlers for NUMERIC_ERROR should handle this exception, since it is the first handler for the predefined exception called NUMERIC_ERROR.

4. Check that an exception declared in a recursive procedure, unlike variables declared in a recursive procedure, is not replicated anew for each recursive activation of the procedure, i.e., different recursive activations should all refer to "the same" declared exception. Check that this holds also for exceptions declared in blocks inside recursively called procedures.

5. Check that exceptions declared in generic packages or procedures are considered distinct for each instantiation.

6. Check that an exception declared in a task body is not replicated for each initiation of the task, i.e., even if several instances of the task body are concurrently active, they all share the same exception.

Gaps

1. It is not stated in the LRM that if the MACHINE_OVERFLOW attribute is true for a type, NUMERIC_ERROR will be raised when overflow is detected and if MACHINE_OVERFLOW is false, NUMERIC_ERROR will never be raised for overflow, although it may presumably (?) still be raised for division by zero (see 13.7.1).

2. The situations under which STORAGE_ERROR is raised are not adequately described here or in 11.7.

11.2 Exception Handlers

Semantic Ramifications

When an exception handler is entered, variables that are being held in registers need not be stored home. In addition, some freedom is permitted regarding when an exception may be raised (see 11.8).

Compile-time Constraints

1. The exception name mentioned after the word when, if not predefined, must have been declared as an exception.

2. A set of exception_choices must not refer to an exception more than once, either within a given exception-handler or in a sequence of exception-handlers.

3. An others exception choice if present, must appear as the only choice in the last handler of the sequence of handlers.

Test Objectives and Design Guidelines

These exception handlers should be designed so as not to raise any exceptions, either explicitly or implicitly. In particular, this means that the exception being handled is not to be re-raised.

Note: Single level exception handling for all predefined exceptions will be checked implicitly as a result of tests for constructs raising predefined exceptions. These tests are associated with the LRM sections where the constructs are defined. However, an additional test is defined in 11.4.1.T/4.

1. Check that the name in a handler must be an exception (predefined or programmer-defined). (Use of the notation STANDARD.NUMERIC_ERROR, etc. in a handler is tested in 8.3.g.T/7. Checks that a declared exception name is or is not hidden appropriately are performed in 8.3.?.)

2. Check that an exception cannot be referred to more than once in a single handler or a sequence of handlers. Try names of the form STANDARD.NUMERIC_ERROR and NUMERIC_ERROR as well as P.SINGULAR and SINGULAR (where P is a procedure and SINGULAR is a locally defined exception).

3. Check that others can only appear by itself at the end of a sequence of exception-handlers and that more than one others choice is not allowed.

4. Check that a

   - predefined exception, or a
   - programmer-defined exception

   raised several (at least 3) levels inside a hierarchy of nested blocks can be successfully handled in an outer block several levels away from the place of occurrence. Handlers:

   - containing a single exception_choice;
   - several exception_choices; and
   - a non-specific handler (others)

should be tried. The handler should not itself raise any exception. One or two other exception handlers, including one mentioning the exception by name, should be present in some block textually enclosing the first exception-handling block.

Implementation Guideline: Try nested blocks both inside and outside an exception handler.

5. Check that a return statement can appear in an exception handler and that it causes control to leave the subprogram containing the handler.

   Implementation Guideline: Check both functions and procedures.

6. Check that local variables (and parameters) of a subprogram, package, or task are accessible within a handler.

7. Check that an exit statement in a handler can transfer control out of a loop.

11.3 Raise Statements

Compile-time Constraints

1. The innermost construct enclosing a non-specific raise statement (i.e., a raise statement not mentioning any exception name) must be an exception handler, not a subprogram, package, or task body.

2. The identifier mentioned in a raise statement, if not predefined as an exception, must have already been declared as an exception (and be within the scope of that declaration).

Test Objectives and Design Guidelines

1. Check that the identifier mentioned in a raise statement must be an exception (predefined or programmer-defined).

   Implementation Guideline: Redeclare the predefined exception names as non-exceptions, e.g. INTEGER, and then try to use the redeclared names in a raise statement. In addition, try a programmer defined integer variable name and undeclared names in an exception handler.

2. Check that a non-specific raise statement found outside an exception handler is forbidden.

   Implementation Guideline: In addition to a simple cases, try raise statements in contexts like the following:

```
when E1|E2  =>
   declare
      procedure P is
      begin
         raise; -- illegal
      end P;
   begin
      raise RANGE_ERROR;
   exception
      when RANGE_ERROR => P;
   end;
```

3. Check that the non-specific raise statement propagates the exception being handled to another handler; use the raise statement in

   - a handler specific to that exception;

   - a handler for several exceptions including the one being propagated;

   - a non-specific handler.

4. Check that when an inner unit redeclares an exception name (either predefined or programmer-defined - two cases to check), thereby hiding the definition previously in effect, the hidden definition is still available for use by component selection (from the outer module, or, respectively, from the predefined module STANDARD).

   Implementation Guideline: The outer unit must possess a name. This means that the unit must be a procedure, function, package, task, or named block.

   Implementation Guideline: This objective can be achieved by repeating the test for 11.1.T/3, with the difference that the innermost of the 3 blocks under consideration there should contain not only a handler for NUMERIC_ERROR but also a handler for STANDARD.NUMERIC_ERROR.

5. Check that a statement of the form raise [exception_name] when condition; is forbidden.

11.4 Dynamic Association of Handlers With Exceptions

Semantic Ramifications

Both 11.4.1 and 11.4.2 are discussed here.

Note that rule (f) of 11.4.1 does not apply directly if an exception is raised in the declarative part of a nested block; this situation is covered by rule (b) of 11.4.2, which states that such an exception is propagated from the block. Since the block is one of the handler's sequence of statements, rule 11.4.1(f) applies to the propagated exception.

It is possible to propagate exceptions out of compilation units. If the exception is raised during elaboration of the compilation unit, the exception is propagated according to the usual rule, i.e., to the context statically enclosing the compilation unit (see below, for example). For library units (see 10.1), there is no statically enclosing context, and hence whether the exception is propagated to the environment enclosing the main program is outside the purview of the language, and so is implementation dependent. (To help in supporting debugging environments, an implemntation should preserve the contents of the stack (if there is one) until a handler for a propagated exception has been found.) For subunits, the exception is propagated to the environment containing the subunit's stub. Hence, if we have:

```
procedure P is
   package Q is
      ...
   end Q;
   package body Q is separate;
begin ... end P;
```

and an exception is raised during elaboration of Q's body, the exception will be propagated to P's caller, just as would be the case for any exception raised in P's declarative_part. (P need not be the main program.) Of course, if P were a package instead of a procedure, propagation of the exception is implementation dependent.

Elaboration of a subprogram consists of elaborating the subprogram's specification. Moreover, subprogram parameters and return types are elaborated only once, the first time they are seen, no matter how many times the complete specification is subsequently repeated. Consequently, elaboration (as opposed to invocation) of a subunit that is a procedure or function can never raise an exception, since exceptions can only be raised when parameters and return types are elaborated, and these will have been elaborated earlier. For example, consider:

```
with Q;
package P is
   function F(X: WEEKDAY range Q.M..Q.N) return INTEGER;
-- RANGE_ERROR raised if Q.M..Q.N is non-null and either
--      bound is not a WEEKDAY value
end P;

with Q;
package body P is
   function G(X: WEEKDAY range Q.M..Q.N) return INTEGER;
   function F(X: WEEKDAY range Q.M..Q.N) return INTEGER is
         begin ... end F;
   function H(X: WEEKDAY range Q.N..Q.M) return INTEGER is
         separate;
end P;
```

In the package body, no exception is raised when elaborating F since F's parameter has already been elaborated. However, an exception can be raised

when G is elaborated, since this is the first occurrence of G's specification. Similarly, an exception could be raised when elaborating H's specification, but no exception would be raised when elaborating H's body (which is a separately compiled subunit), since H's parameter will already have been elaborated once.

Exceptions, unlike variables, have an existence outside the scope of their declaration. In particular, they can be propagated into scopes where they cannot be named, and in such cases, they can be handled only by an others handler. Moreover, it is possible to propagate an exception out of scope and then back into scope (see below). The practical consequence is that all declared exceptions must be uniquely identifiable at run time independent of the scope in which they are declared. For example, consider the following set of packages and procedures. They have been designed so the exception Q.ECP is propagated out of scope and then back into scope. The idea is that Q.G calls P.F, which calls Q.H, which raises exception Q.ECP. Since ECP is local to Q and not known to P, it is propagated out of scope to F and back into scope to G.

```
package P is
   procedure F;      -- called by Q.G
end P;

package Q is
   procedure G;      -- call this procedure first
   procedure H;      -- called by P.F
end Q;

with (P)
package body Q is
   ECP : exception;
   procedure G is
   begin
      P.F;       -- call F;
   exception
      when ECP => ...     -- Q.ECP can be handled here,
                          -- even after it is propagated out of
                          --     scope
   end G;

   procedure H is
   begin
      raise ECP;          -- propagated to H's caller
   end H;
begin
   Q.G;                   -- start execution
end Q;

with Q;
package body P is
   ECP : exception;        -- distinct from Q.ECP
   procedure F is
```

```
      begin
         Q.H;               -- call H
         raise ECP;         -- raise P.ECP (won't be executed)
      exception
         when ECP => ...    -- only handles P.ECP, not Q.ECP
         when others => raise;      -- handles Q.ECP
      end F;
   end P;
```

Note that all the above units can be compiled separately or together (without the with clauses) without affecting the propagation and handling of Q.ECP.

Test Objectives and Design Guidelines

1. Check that any exception raised in a subprogram body is propagated to callers of the subprogram, not to the statically enclosing lexical environment.

   Implementation Guideline: Both predefined and programmer-defined exceptions should be checked.

   Implementation Guideline: Some of the subprograms in the dynamic call chain should have exception handlers and some should not. Automatic propagation through more than one level of call should be attempted in some test cases.

2. Check that an exception raised during elaboration of a subprogram's declarative part is propagated to the caller of the subprogram, not the environment statically enclosing the subprogram's body.

   Implementation Guideline: The subprogram body should contain a handler for the exception raised in the declarative part (although this handler should not be executed).

   Implementation Guideline: All predefined exceptions except SELECT_ERROR and TASKING_ERROR can occur when declarative parts are elaborated. The compile-time processing of initializations and checking of constraints need not be performed by the same compiler code that processes assignment statements. However, we assume that it would be extremely unlikely for a compiler to propagate one exception correctly and not another. Therefore only the following 3 cases need be included in the tests:

   - a predefined exception propagated from a function called when elaborating the declarative part;

   - a predefined exception raised when elaborating a constraint and by initialization;

   - a programmer-defined exception propagated out of a function called when elaborating the declarative part.

   Assumption: A compiler correctly propagating a predefined exception

occurring during elaboration of declarations will correctly propagate every predefined exception in the language; likewise, a compiler correctly propagating one programmer-defined exception will correctly propagate any programmer-defined exception.

3. Check that exceptions raised during elaboration of package specifications, task specifications, or declarative_parts of blocks and package bodies are propagated to the environment statically enclosing the block, package, or task (note that no exceptions are propagated from a task body; see 11.4.1(d)). As a subcase, check specifically that exceptions raised by functions invoked during elaboration of a declarative part are always propagated outside the declarative part.

   Implementation Guideline: The block or package should have a handler for the exception being propagated (although this handler should not be executed).

4. Check that

   - an implicitly-raised exception,
   - an explicitly-raised predefined exception,
   - an explicitly-raised programmer-defined exception, and
   - a programmer-redeclared predefined exception

   occurring in the body of a unit possessing:

   - a handler specific to the exception,
   - a handler for several exceptions ("A or B or C") including the one that occurred, and
   - a non-specific handler (others)

   is always handled locally.

5. Check that exceptions propagated out of a handler are propagated outside the unit containing the handler, without invoking some other handler associated with that unit or recursively re-invoking the same handler.

   Implementation Guideline: There should be no blocks inside these handlers.

6. Check that exceptions occurring in a block declared inside an exception handler follow the rules for blocks.

7. Check that the statement part of package bodies can raise, propagate, and handle exceptions. Check also hat if a package body's exception handler does not raise an unhandled exception, no exception is propagated out of the package body.

8. Check that "systematic unwinding" is possible: an exception occurring deep inside a combined static/dynamic hierarchy is handled successively by each unit in the propagation path, each handler on the path performing its unit's "last wishes" and re-raising the exception to allow the remaining units on the propagation path to perform <u>their</u> "last wishes".

   <u>Implementation Guideline</u>: Use blocks, packages, and subprograms.

9. With a programmer-defined exception, check that the exception can:

   - be handled out of scope only with a non-specific handler, then be re-raised for dynamic propagation back into its original scope; and, finally,
   - be handled again, back in scope, under its original name.

   Check also that predefined exception names redeclared by the programmer behave similarly (with the difference that out-of-scope a specific handler for that name refers to the predefined exception).

   <u>Implementation Guideline</u>: Check using separately compiled units as well as single units.

10. Check that if an exception is raised during elaboration of a subprogram specification contained in a package specification or package body:

   - the exception is not handled within the body of the subprogram;
   - the exception is not handled by a handler in the package body;
   - if the subprogram specification appears in the package specification, the exception is raised and handled only in the context of the package specification, not in the context of the package body;
   - if the subprogram specification appears only in the package body, the exception is handled only in the context of the package body.

<u>11.5 Exceptions Raised During Task Communication</u>

<u>**Exceptions**</u>

1. **If task S calls an entry in task T, TASKING_ERROR is raised in S at the entry call if T is terminated at the time of the call or prior to accepting the call. (This check cannot be suppressed.)**

2. **During a rendevous, TASKING_ERROR is raised in the calling task if the task containing the accept statement is abnormally terminated as the result of an abort statement or as the result of an unhandled FAILURE exception received during the rendevous (see 11.5). (This check cannot be suppressed.)**

Test Objectives and Design Guidelines

1. If a user-defined or predefined exception (other than FAILURE) is raised during a rendevous and not handled within the rendevous, check that the exception is propagated both within the calling task (at the point of the entry call) and within the called task at the point of the accept statement.

2. Check that TASKING_ERROR is raised under the appropriate conditions:

   - when the called task is terminated at the time of the call (TASKING_ERROR is raised in the calling task);
   - when the called task is not terminated at the time of the call, but terminates before the entry call is accepted (TASKING_ERROR is raised in the calling task);
   - when TASKING_ERROR is raised explicitly or by propagation within the accept statement (TASKING_ERROR is raised in both tasks);
   - when the called task is terminated by an abort statement during rendevous (TASKING_ERROR is raised in the calling task);
   - when the called task is terminated by FAILURE during a rendevous (TASKING_ERROR is raised only in the calling task (see also 11.6)).

11.7 Suppressing Exceptions

Semantic Ramifications

It is not clear whether

```
pragma SUPPRESS(OVERFLOW_CHECK, A);
```

means that an overflow check is or is not to be performed for A + B. Similarly, if DISCRIMINANT_CHECK is suppressed for object C.D, where D is a component of a variant part, does this mean that only the discriminant checks for accessing component D are to be suppressed or that the checks on accessing components of C.D are to be suppressed, or both? Additional questions are listed under Gaps. Since pragmas can, in any event, be ignored, these questions are not as important in evaluating the validity of a compiler as would otherwise be the case. However, they do serve to indicate that unless the intended semantics of an obeyed SUPPRESS pragma are more clearly specified, implementations will give unnecessarily divergent interpretations.

Since a SUPPRESS pragma need not be obeyed, the only conformity checks we indicate at this time are those needed to check that the pragma is recognized properly. Checks to evaluate whether SUPPRESS is being obeyed will be specified in later versions of the Implementers' Guide. The currently specified constraints and tests indicate the minimum support every implementation must provide for SUPPRESS.

Compile-time Constraints

1. The check_name appearing as the first argument of SUPPRESS must be one of the predefined names (see T/2 below).

2. The second argument of the SUPPRESS pragma must not be the name of a number, enumeration literal, statement label, block or loop identifier, attribute, function call, subprogram, entry or member of an entry family, exception, slice, or operator_symbol.

3. The pragma can only appear in a declarative part of a block or the body of a subprogram, package, or task.

Test Objectives and Design Guidelines

1. Check that an identifier ending in _CHECK but not one of the predefined check names cannot be given as the first argument of SUPPRESS.

   Check that the second argument of SUPPRESS cannot be one of the names listed in the second constraint above.

   Check that SUPPRESS is not permitted in a package specification or task specification, or in the statement part of a unit, or immediately before or after the context_specification of a compilation unit.

2. Check that any of the predefined check_names are permitted as the first argument of SUPPRESS, namely, names beginning with ACCESS, DISCRIMINANT, INDEX, LENGTH, RANGE, DIVISION, OVERFLOW, and STORAGE and ending with _CHECK.

   Implementation Guideline: Try a SUPPRESS pragma even when objects and types have been declared with one of these check_names.

3. Check that the second argument of SUPPRESS can be an object name (including the name of an object component, e.g., A(I)), a type name, or a subtype name (including task type names).

   Check that the form of SUPPRESS with ON is accepted.

Gaps

1. What does it means to suppress OVERFLOW_CHECK or DIVISION_CHECK for a numeric variable? for a non-numeric variable? Can LENGTH_CHECK be suppressed for non-array objects or types? In general, is it intended that one can specify suppression of checks only for objects and types to which the specified check applies?

2. What is the meaning of suppressing a check for a component of an object?

3. Presumably if a component name is specified, it is elaborated when the SUPPRESS pragma is elaborated, but nothing is explicitly said about when this pragma is elaborated. Presumably it is elaborated prior to any

following declarations and after any preceding declarations, and elaboration of the object name or type name means establishing the identity of the named entity.

4. The situations under which each check is performed are not precisely specified, although we have attempted to indicate with each exception condition listed in this document what check_name suppresses the check.

11.8 Exceptions and Optimizations

Semantic Ramifications

The purpose of this section of the LRM is to indicate that certain optimizations can be provided by an implementation. One set of optimizations permits evaluating a function earlier than its textual position would indicate, as long as the function would eventually be evaluated before control leaves a certain region of text, namely the statements between <u>begin</u> (or <u>do</u>) and <u>exception</u> or <u>end</u>. In short, a certain amount of imprecision is allowed with respect to when an exception is raised in such a sequence of statements. The rules allow evaluation of arithmetic expressions in parallel or with pipelined arithmetic units even if the existence of NUMERIC_ERROR is not indicated until control has entered some statement following an arithmetic expression. The ony requirement is that no exception due to evaluating some operation be raised after control has passed out of the sequence of statements bounded by <u>begin</u> (or <u>do</u>) and <u>exception</u> or <u>end</u>, or into an inner block, body, or accept statement's sequence of statements (see Gaps). Hence, if a programmer requires more precision over the region of text where a particular exception can be raised, he can enclose the desired text in a block.

The rules are written primarily with the predefined operations in mind, but they are stated so they apply to user-defined functions as well if a compiler can determine that they are functions satisfying the criteria listed in the first paragraph of 11.8.

Since an implementation can suppress the evalution of operations yielding unneeded values, boolean expressions can always be short-circuit evaluated if no user-defined subprograms are invoked, e.g.,

```
if TRUE or A = B(50) then
```

need not raise CONSTRAINT_ERROR even if B'LAST is less than 50. On the other hand, given

```
begin
    D := 30;
    if A = D/(A-3) or A = B(50) then ...
exception
```

there is no guarantee that NUMERIC_ERROR will be raised instead of CONSTRAINT_ERROR if A = 3. Either operand of <u>or</u> can be evaluated first. Moreover, if CONSTRAINT_ERROR is raised, there is no guarantee that D = 30 in

a handler for this exception, since B(50) can be evaluated before the assignment to D is performed. However, given

```
begin
    D := 30;
    if A = D/(A-3) or else A = B(50) then ...
exception
```

B(50) cannot be evaluated before D/(A-3) has been evaluated, and hence, if CONSTRAINT_ERROR is raised, D must equal 30.

These examples illustrate some of the consequences of the rules in 11.8, both for the programmer and the implementer.

Test Objectives and Design Guidelines

Since an implementation need not provide any of the optimizations permitted by this section, we do not give any test objectives for this section in this version of the IG. Later versions will contain test objectives to check that the permitted optimizations are not applied inappropriately.

Gaps

1. The first bullet in the LRM should say that whether or not an exception-raising operation's value is needed, it must not be invoked after control has left the sequence of statements containing the operation. Otherwise, the invocation could cause an exception to be handled by a different handler than the one that would have been invoked if the "optimization" had not been performed. It should also be stated that the exception resulting from any moved operation must not be raised while executing an inner block, body, or accept statement. In essence, the statements of a block, body, or accept statement must act as barriers across which an exception-raising operation cannot be moved.

CHAPTER 12

Generic Program Units

12.1 Generic Declarations

Semantic Ramifications

Note that in a generic subprogram/package declaration/body there are no additional constraints that certain expressions have static values beyond those static expresssion constraints that apply to any (nongeneric) subprogram/package.

In so far as possible. expressions appearing in a generic_part must be evaluated during the elaboration of the generic_part, which is part of the elaboration of a generic declaration. However, primaries that depend on an (earlier) generic formal type parameter cannot be evaluated until the corresponding actual parameter is available, i.e., during the elaboration of a generic instantiation. Thus, operations, all of whose operands are free of generic formal parameters, must be evaluated during the generic declaration elaboration. whereas operations, some of whose operands contain generic formal parameters, must be evaluated during the generic instantiation elaboration.

Consider, for example:

```
generic
    type T is range <>;
    K : INTEGER := (A + B) + INTEGER(T'LAST) + C;
procedure P ...;
```

The primaries A. B, and C, and the subexpression (A + B) must be evaluated during the elaboration of the generic declaration. The primary T'LAST, the conversion to INTEGER. and the surrounding two additions must be evaluated during each instantiation of P (using the appropriate actual type parameter for T).

Note that if the leftmost addition (A + B) were to raise an exception (NUMERIC_ERROR), then the exception must be propagated into the context containing the generic declaration. whereas the other additions must propagate their exceptions into the context containing the elaborated generic instantiation.

The LRM appears to allow variables to be used in parameter declarations in a generic part; however, this may be an oversight (see Gaps 12.1.G/1).

Note that even though the identifier of a generic subprogram/package textually occurs after the generic part in a generic declaration, the identifier is introduced prior to elaborating the generic part, and thus any outer declaration of the identifier is hidden (or overloaded) (see LRM 8.2 and 8.3). For example:

```
package P is
    type T is (A, B, C);

    generic
        Q : INTEGER := T'SIZE;     -- size of the enumeration type T
        R : INTEGER := P.T'SIZE;   -- scope error. inner P, but T not
                                   -- yet introduced
        type T is range <>;
        S : INTEGER := P.T'SIZE;   -- size of the integer type T
    package P is
        ...
    end P;
end P;
```

Note that a generic formal type may be used as the type in a subsequent generic formal (object) parameter declaration in the same generic part, as in:

```
generic
    type T is range <>;
    I : T := 0;
```

Note that a generic subprogram/package has two templates established for it. The elaboration of a generic subprogram/package declaration establishes a template subprogram/package specification. The elaboration of a generic subprogram/package body establishes a template subprogram/package body (LRM 12.2). Establishing a template includes identifying (i.e., binding) all uses of names in the template according to the scope and visibility rules in LRM 8.2 and 8.3. Thus, according to LRM 12.3, generic subprograms/packages follow the same static binding rules as nongeneric subprograms/packages with respect to instantiations and calls. But note that since a generic declaration is textually separate from its corresponding generic body, the body can have a different context (i.e., a different set of bindings for its free identifiers). Each template is bound in the context in which it textually occurs (see 12.1.G/4).

<u>Compile-time Constraints</u>

1. A generic formal parameter P1 may only be referred to by another generic formal parameter P2 of the same generic part if both P1 is a type and P1 appears (is "declared") before P2.

2. Neither a choice, nor an integer type definition, nor an accuracy constraint, may depend on a generic formal parameter. This applies to the generic part as well as to the specification and body of a generic subprogram/package. (See 12.1.G/3.)

3. **The generic formal parameter identifiers of a generic part must be distinct from each other, and either from the identifiers declared in the generic subprogram's formal and declarative parts, or from the identifiers declared in the generic package's specification and the body's declarative part (see LRM 8.3).**

4. A generic subprogram must not be used as the called subprogram in a subprogram call (i.e.. only subprogram instantiations can be called).

5. The name of a generic package must not be used in use_clauses nor as the package name in selected components (LRM 4.1.3/c) outside of the generic package (i.e., only the name of an instantiation of a generic package can be so used, see 12.1.G/5). (However. 4.1.3/e does apply within a generic package.)

Test Objectives and Design Guidelines

Note that in order to check that a generic declaration has a certain property, it may be necessary to instantiate it, or even to invoke the instantiated unit.

1. Check that:

   - a generic formal parameter must not refer to a later generic formal parameter (neither a type nor an object) of the same generic part.

   - A generic formal parameter must not refer to an earlier generic formal object/subprogram parameter of the same generic part.

2. Check that neither a choice, nor an integer type definition, nor an accuracy constraint, may depend on a generic formal parameter.

   Implementation Guideline: Try these in generic parts as well as in specifications and bodies of generic subprograms/packages.

3. Check that (sub)expressions appearing in a generic part and having no primaries that depend on generic formal type parameters are evaluated during the elaboration of the generic part. Likewise, check that the remaining (sub)expressions in the generic part, which have primaries that depend on generic formal type parameters, are evaluated during the elaboration of each generic instantiation of the generic subprogram/package.

   Implementation Guideline: Cause the (sub)expression in question to raise an exception and have separate handlers, one for the generic declaration and one for the generic instantiation.

4. Check that the elaboration of a generic declaration does not elaborate the subprogram/package specification portion of the declaration.

   Implementation Guideline: Have the expressions in the specification portion raise exceptions if they are evaluated.

5. Check that the identifier of a generic subprogram/package is introduced prior to the elaboration of the generic part and that any outer declaration of the identifier is appropriately hidden (or overloaded). Try both legal and illegal cases, as in the example in 12.1.S above, but in separate tests.

6. Check that a generic formal type may be used as the type mark in a subsequent generic formal (object) parameter declaration in the same generic part (e.g., as in the example in 12.1.S above).

   Check that any additional constraints explicitly specified after the type mark (in the formal object parameter declaration) are compatible with the corresponding generic actual type parameter's constraints in a generic instantiation.

   Repeat the above two checks, but with the generic formal type parameter used in other generic formal type parameters and in generic formal subprogram parameters.

7. Check that the generic formal parameters of a generic part are elaborated in linear (textual) order.

   Implementation Guideline: Use functions with global side effects (but see 12.1.G/1), otherwise use exceptions.

8. Check that the sequence of generic formal parameters in a generic part cannot be enclosed in parentheses.

   Check that the generic formal parameters in a generic part cannot be separated by commas (instead of semicolons).

   Check that each (especially the last) generic formal parameter in a generic part must end with a semicolon.

9. Check that 12.1.C/3 is satisfied (see ?).

10. Check that the names in a generic subprogram/package declaration/body are statically identified (i.e., bound) at the point where the generic declaration/body textually occurs, and are not dynamically bound at the point of instantiation.

    Implementation Guideline: Include a test where a generic declaration and its corresponding body contain the same free identifiers, but are in different contexts, and thus have different bindings for the free identifiers.

11. Check that a generic subprogram (i.e., the template) cannot be used as the called subprogram in a subprogram call (i.e., an instantiation should be used).

    Implementation Guideline: Instantiate the generic subprogram first, but then call the template instead of the instantiated instance. Don't have any generic formal parameters in the generic part. Have the call's actual parameters match the nongeneric formal parameters of the generic subprogram's specification.

12. Check that the name of a generic package (i.e., the template) cannot be used in use_clauses, nor as the package name in selected components

outside of the generic package (i.e., the name of an instantiation should be used).

Implementation Guideline: Instantiate the generic package first, but then use the template name instead of the instantiated name, first, in a selected component form of name, and then, in a use_clause. Don't have any generic formal parameters in the generic part.

Gaps

1. Section 6.1 of the LRM prohibits names of variables and calls to user-defined operators, functions, and allocators from appearing in expressions in the formal_part (i.e., the parameter declarations) of a subprogram specification. However, there seems to be no similar restriction for discriminant_parts (LRM 3.7.1) nor for generic_parts (LRM 12.1). LRM 12.1.1 states that: "The usual forms of parameter declarations available for subprogram specifications can also appear in generic parts." It is not clear whether or not the above restriction also applies. If it does apply, then should it also apply to generic formal type parameters and generic formal subprogam parameters in generic parts, and to discriminant parts? Note that the reasons for the restriction (namely, that the specification for a subprogram may textually occur more than once, and that the restriction forces the elaboration of each occurrence to yield the same expression values for constraints, etc.) applies to discriminant parts (for incomplete type declarations, LRM 3.8), but not to generic parts (which can only occur once, see 12.2.S).

2. It is not clear when a generic subprogram declaration hides or overloads another subprogram declaration of the same name. The rules in LRM 8.3 and 6.6 refer only to the subprogram specification, which can contain generic formal types. The generic part is not mentioned at all in these sections. These rules clearly apply to the particular subprogram resulting from an instantiation, but what about the generic declaration itself?

3. Constraint 12.1.C/2 should probably be extended to say that in any construct that requires a static expression, the expression must not depend on a generic formal parameter. This would include representation specifications.

4. The last sentence in the Note in LRM 12.1 should read: "When a template [specification (or body)] is established, all names occurring within it must be identified in the context of the generic declaration [(or body)]." Actually, this sentence should be part of the semantic rules of 12.1 instead of a note.

5. LRM 4.1.3/c, 8.3/b, and 8.4 are not completely clear that the use of the word "package" means nongeneric packages and instantiations of generic packages, but not generic packages. However, the examples in LRM 12.4 show that the intent is 12.1.C/5.

12.1.1 Parameter Declarations in Generic Parts

Semantic Ramifications

Note that the value of a generic formal in parameter is always a copy of the value provided by the corresponding generic actual parameter in a generic instantiation, whereas a nongeneric formal in parameter that has an array, record, or private type need not be a copy of the corresponding actual parameter (see LRM 6.2).

Note that a generic formal in out parameter always acts as an object name renaming (as in a renaming declaration, see LRM 8.5) the corresponding generic actual parameter supplied in a generic instantiation, whereas a nongeneric formal in out parameter that has a scalar or access type is always a local copy of (and that has an array, record, or private type is allowed to be a local copy of) the corresponding actual parameter (see LRM 6.2). Note that the constraint about the generic actual parameter not being a potentially nonexistent component of an unconstrained object with discriminants (12.1.1.C/5) is the same constraint as when renaming the component in a renaming declaration (see LRM 8.5). (This constraint really belongs in LRM 12.3.1.)

Compile-time Constraints

1. **Parameter declarations in generic parts must have the modes in or in out, and must not have the mode out.**

2. **Formal in out parameter declarations (also in parameters with nonassignable types, see LRM 7.4.2) in generic parts must not have initializations.**

3. **A generic formal in parameter must not be used as an actual in out or out parameter, nor as the target of an assignment (see LRM 6.2).**

4. **A generic actual in out parameter must be a variable and be of a type for which assignment is available (see 12.1.1.G/1).**

5. **A generic actual in out parameter must not be a component of an unconstrained object having discriminants when the existence of the component depends on the value of a discriminant.**

Exceptions

1. **CONSTRAINT_ERROR is raised if the value of the initialization expression in a parameter declaration in a generic part does not satisfy the constraints of the subtype indication of the parameter declaration (but see 12.1.1.G/2).**

Test Objectives and Design Guidelines

1. **Check that parameter declarations in generic parts cannot have the mode out.**

2. Check that generic formal in out parameter declarations cannot have initializations.

   Check that generic formal in parameter declarations with nonassignable types (in particular, limited private, task, and composite types containing such component types) cannot have initializations.

3. Check that a generic formal in parameter cannot be used as an actual (nongeneric) out parameter, nor as an actual (generic or nongeneric) in out parameter, nor as the target of an assignment.

   Implementation Guideline: Use a scalar, an array, a record, and an access type as in parameters. For the array and record types, also try altering their components as well as the entire objects.

4. Check 12.1.1.E/1 as in 6.1.T/8.

5. Check that a generic formal in parameter is always a copy of the actual parameter value.

   Check that a generic formal in out parameter is always a renaming of the actual parameter variable.

   Implementation Guideline: Use a scalar, an access, an array, a record, and a private type as the parameter types. Also see the guidelines for 6.2.T/3.

6. Check that mode in is assumed when no mode is explicitly specified in a parameter declaration in a generic part.

7. Check 12.1.1.C/5.

Gaps

1. The sentence in LRM 12.1.1 that states constraint 12.1.1.C/4 above appears to us to be a mistake. We have not found any justification for the assignable type portion of it, and believe that the sentence and the constraint should be deleted. (The variable-only portion is already covered in LRM 12.3.1.) The result would then be that generic in out parameters have the same constraints as object renaming declarations (LRM 8.5), which are slightly more severe than for ordinary subprogram in out parameters (LRM 6.2). Neither a renamed object nor a subprogram actual in out parameter need be of an assignable type (thus allowing limited private objects to be renamed and passed as in out parameters (LRM 7.4.2)). Note that if the type of a generic formal in out parameter is an ordinary type (i.e., not a generic formal type), then LRM 12.3.1 requires the corresponding actual parameter to have the same type (which is known at compile time), and so the legality of assignment to the formal parameter within the generic subprogram/package body can be determined when the generic body is compiled (into a template). Similarly, if the type of a generic formal in out parameter is a generic formal type, then LRM 12.1.2 states which operations can be assumed to be available within the generic

(specification and) body, depending on the type's class (in particular, assignment for all but the limited private class). The matching rules in LRM 12.3 require the corresponding generic actual type to have at least the assumed operations, which is checkable at compile time.

2. LRM 12.1.1 implies that generic formal in parameters are similar to nongeneric formal in parameters. LRM 6.1 says that a nongeneric in parameter's default value is evaluated and checked against the parameter's subtype constraints when the subprogram specification is elaborated. This is possible because 6.1 prohibits the default value from depending on other formal parameters in the same formal part. But this dependence is allowed for generic formal parameters (see 12.1.S). Clearly, when such a dependence occurs, the constraint checking must occur during the elaboration of a generic instantiation. But when a generic formal in parameter does not depend on other generic formal parameters, the LRM is not clear as to whether the constraint checking of a default value occurs when the generic declaration is elaborated or when a generic instantiation is elaborated.

12.1.2 Generic Type Definitions

Semantic Ramifications

Note that the elaboration of a generic type definition immediately makes available certain predefined operations (depending on the type class), which can be used in subsequent generic formal parameters of the same generic part (but see 12.1.2.G/1). As subsequent generic formal subprogram parameters (that use the generic type) are elaborated, they successively increase the set of available operations, according to the linear elaboration order of generic parts (LRM 12.1) and the scope and visibility rules (LRM 8.2, 8.3).

A generic array type definition makes available the usual predefined operations for arrays, which include assignment, (in)equality, indexing, slicing, attributes, index constraint notation, aggregate notation, concatenation (for one-dimensional arrays), logical (for BOOLEAN arrays), relational (for one-dimensional discrete arrays), membership, conversion, and qualification.

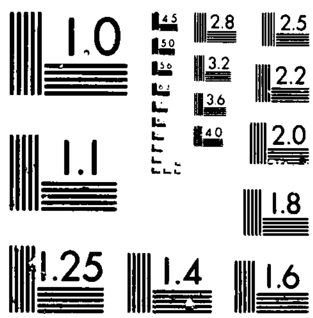
A generic access type definition makes available the usual predefined operations for access objects, which include assignment, (in)equality, implicit dereferencing (combined with indexing, slicing, or selection, depending on the accessed type), explicit dereferencing (.all, attributes, index/discriminant constraint notation (depending on the accessed type), allocation, membership, conversion, and qualification.

A generic (limited or nonlimited) private type definition makes available the usual predefined operations for (limited or nonlimited) private objects, which include assignment (for nonlimited private types only), (in)equality (for nonlimited private types only), selection (of discriminants, if any), attributes, discriminant constraint notation (if any), membership, conversion, and qualification (but see 12.1.2.G/2).

AD-A091 760   SOFTECH INC WALTHAM MASS   F/G 9/2
ADA COMPILER VALIDATION IMPLEMENTERS' GUIDE.(U)
OCT 80 J B GOODENOUGH   MDA903-79-C-0687
UNCLASSIFIED 1067-2.3   NL

4 OF 4
AD
A091760

END
DATE
FILMED
12-80
DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

A generic scalar type definition ((<>), range <>, digits <>, or delta <>) makes available the usual predefined operations for objects of the corresponding type class (discrete, integer, floating point, or fixed point), which include assignment, (in)equality, attributes (except IMAGE and VALUE), range and/or accuracy constraint notation, literal notation, the appropriate predefined (logical, relational, adding, unary, multiplying, and exponentiating) operators, the function ABS, membership, conversion, and qualification. Note that a generic integer type definition makes available the predefined "*", "/", and "**" operators for which the left operand (or the right, for "*") is of a fixed point type and the other operand is of the generic integer type. Similarly, the floating point "**" is made available. Similar remarks apply to generic floating and fixed point type definitions.

Compile-time Constraints

1. A generic formal type parameter must not have a discriminant part if its generic type definition is not a private type definition.

2. The base type of a generic formal type must not be used as the component/accessed base type in the corresponding generic array/access type definition.

Exceptions

Exceptions can be raised during the elaboration of discriminant parts (3.7.1), array type definitions (3.6), and access type definitions (3.8).

Test Objectives and Design Guidelines

1. Check that a generic formal type parameter with a generic scalar, array, or access type definition cannot have a discriminant part.

2. Check that a generic formal type parameter cannot have a record type definition.

3. Check that the base type of a generic formal type cannot be used as the component/accessed base type in the corresponding generic array/access type definition.

Gaps

1. LRM 12.1.2 says the operations available for a generic type are available in the specification and body of a generic subprogram/package. It doesn't mention availability within the generic part. However, LRM 12.1, 8.2, and 8.3 imply to us that the operations are also available in the generic part, as discussed in 12.1.2.S above.

2. LRM 12.1.2 says that for a generic formal limited private type, no operation is available. This appears to be an oversight. We believe that at least the operations listed above in 12.1.2.S are available.

12.1.3 Generic Formal Subprograms

Gaps

1. In the case where a generic formal subprogram parameter contains a default name, it is not clear in which context (the generic declaration or a generic instantiation) the identity of the default subprogram is to be determined. The problem is similar to 12.1.1.G/2.

12.2 Generic Bodies

Semantic Ramifications

Note that a generic subprogram/package body is never preceded by a (copy of the) generic_part. Thus, whereas a nongeneric subprogram body can also serve as the subprogram declaration (see LRM 6.3), this is never allowed for generic subprograms. A generic subprogram must have a separate declaration (which contains the generic_part) and body.

Note that establishing a template body includes statically binding the names within the generic body in the context of the generic body (see 12.1.S and 12.2.G/1).

Compile-time Constraints

1. The constraints about the textual placement of ordinary (i.e., nongeneric) subprogram/package declarations and bodies also apply to generic subprograms/packages (see 3.9, 6.1, 6.3, 7.1, 7.2, 7.3).

Test Objectives and Design Guidelines

1. Check that a generic subprogram (also, package) body cannot be preceded by a copy of the generic part used in the corresponding generic declaration.

   Check that a subprogram body preceded by a generic part cannot serve as a generic declaration when the declaration is omitted.

2. Check that the elaboration of a generic body has no effect other than to establish the body as the template body to be used for the corresponding instantiations.

   Implementation Guideline: The elaboration of a nongeneric package body can change the values of global variables and/or raise exceptions. Neither of these should occur when a generic package body is elaborated; only when it is instantiated.

3. Check that 12.2.C/1 is satisfied (see ?).

Gaps

1. The last sentence of LRM 12.2 should read: "The only effect of the elaboration of a generic body is to establish this body as the template [body] to be used for the corresponding instantiations."

12.3 Generic Instantiation

Semantic Ramifications

The detailed matching rules for the various kinds of generic parameters are discussed in subsections, as in the LRM. This section covers the remaining aspects of generic instantiations.

Note that the newly declared subprogram/package designator/identifier is introduced before elaborating the generic instantiation, thus hiding or overloading any previous declaration of the designator/identifier. However, the designator/identifier cannot be used within the generic instantiation (but only after the instantiation is elaborated), not even in a selected component form of name. For example:

```
function F (B : BOOLEAN) return INTEGER;
function F (I : INTEGER) return INTEGER;
J : INTEGER;
...
generic
    GI : INTEGER;
    with function GF (I : INTEGER) return INTEGER;
function G (I : INTEGER) return INTEGER;
...
function K is new G(J,F);         -- Legal reference to variable J
                                  -- and integer function F.
function J is new G(J,F);         -- Variable J is hidden,
                                  -- hence illegal since function J
                                  -- is passed to integer GI.
function J is new G(J(3),F);      -- Illegal since function J must be
                                  -- called before it is instantiated.
function J is new G(3,J);         -- Illegal since function J is
                                  -- needed as an actual subprogram
                                  -- parameter before it is
                                  -- instantiated.
function F is new G(F(TRUE),ABS); -- Legal reference to boolean
                                  -- function F (i.e., overloading of
                                  -- F); the outer integer function F
                                  -- is hidden by this new integer
                                  -- function F.
function F is new G(F(3),ABS);    -- Illegal since outer integer
                                  -- function F is hidden by this new
                                  -- F, which is not yet instantiated.
```

Note that the names within a template are statically identified (i.e.,

bound) at the textual point of the generic declaration/body when the template is established (see 12.1.S), and not at the point of instantiation. Likewise, the names in a generic instantiation (in particular, in the generic actual parameters) are statically bound at the textual point of instantiation (in effect, before the actual parameters are substituted for the corresponding generic formal parameters in the specification and body templates, see 12.3.G/1). Although generic subprograms/packages are similar to macros, note that most implementations of macro processors use dynamic name binding (as in LISP or APL), which yields different results from static binding when an actual parameter identifier is the same as a local identifier of the generic subprogam/package and is used in the scope of the local identifier.

Note that the elaboration of a generic instantiation must first complete the elaboration/evaluation of those portions of the generic part that were not elaborated as part of the generic declaration elaboration because they depended on generic formal parameters (i.e., on the actual parameter "values", see 12.1.S). After the generic part is completely elaborated, so that all generic formal parameters have (specified or default) actual "values", then an instance of the specification and body templates can be created, etc., as described in LRM 12.3.

Note that an object declaration may follow a generic subprogram/package instantiation (which creates a body) in a declarative part, whereas an object declaration cannot follow an explicit subprogram/package body in a declarative part (see LRM 3.9).

Compile-time Constraints

1. A generic instantiation must not precede the corresponding generic body being instantiated (nor the corresponding generic declaration).

2. A reference to an instantiation of a generic subprogram/package must not precede nor occur in the instantiation itself.

3. In a generic instantiation, a generic actual parameter must be supplied for each generic formal parameter that has no default given in the corresponding generic part.

4. The number of generic actual parameters in a generic instantiation must not exceed the number of generic formal parameters in the corresponding generic part.

5. All the positional generic actual parameters must precede all the named generic actual parameters in a generic instantiation.

6. A named generic actual parameter must not be given for a generic formal parameter that is already associated with an earlier positional or (same) named generic actual parameter in a generic instantiation.

7. The formal parameter name in a generic association must be identical to that of a generic formal parameter in the corresponding generic part.

8. In a generic instantiation, generic formal object parameters must only be given actual object parameters, formal type parameters must only be given actual (sub)type parameters, and formal subprogram parameters must only be given actual subprogram (or entry) parameters.

9. The name following <u>new</u> in a generic instantiation must be the name of a generic subprogram/package.

<u>Test Objectives and Design Guidelines</u>

1. Check that a generic instantiation cannot precede the corresponding generic body nor the generic declaration being instantiated.

   Check that a reference to an instantiation of a generic subprogram/package cannot precede nor occur within the instantiation itself.

   Check that an outer declaration of the same identifier as the newly declared identifier of the generic subprogram/package instantiation is appropriately hidden or overloaded, as in the examples in 12.3.S above.

2. Check that in a generic instantiation:

   - a generic actual parameter cannot be omitted if the corresponding generic formal (object or subprogram) parameter has no default.
   - the number of generic actual parameters cannot exceed the number of generic formal parameters in the corresponding generic part.
   - named generic actual parameters cannot precede nor be interleaved with positional generic actual parameters.
   - a positional or named generic actual parameter and another named generic actual parameter cannot be specified for the same generic formal parameter.
   - the formal parameter identifier in a named generic association cannot be different from all of the identifiers of the generic formal parameters in the corresponding generic part.

   <u>Implementation Guideline</u>: When possible, try each of the above with generic instantiations consisting of (1) positional associations only, (2) named associations only, and (3) mixtures of positional and named associations.

3. Check that generic instantiations of the form P(A,,B) are forbidden, even when the omitted generic parameter has a default.

   Check that generic instantiations of the form P(A|B => C) are forbidden, even when generic formal parameters A and B are of the same kind (object, type, or subprogram) and match (same subtype, same kind of generic type definition, or same subprogram specification).

4. Check that regardless of the order of the named generic actual parameters in a generic instantiation, that each is associated with the corresponding generic formal parameter that has the same formal parameter identifier.

5. Check that in a generic instantiation:

   - actual object parameters cannot be passed to formal type parameters nor to formal subprogram parameters.

   - actual type parameters cannot be passed to formal object parameters nor to formal subprogram parameters.

   - actual subprogram parameters cannot be passed to formal object parmeters nor to formal type parameters.

6. Check that the name following new in a generic instantiation must be the name of a generic subprogram/package.

   Implementation Guideline: Use ordinary nongeneric subprogram/package names. Also try names that denote already instantiated subprograms/packages.

7. Check that the names in a generic instantiation are statically identified (i.e., bound) at the textual point of the instantiation, and are bound before being "substituted" for the corresponding generic formal parameters in the specification and body templates.

   Implementation Guideline: Use tests that distinguish between LISP/APL/macro dynamic binding and Algol/Pascal/Ada static binding, such as when a generic actual parameter identifier is identical to a local identifier of a template and is used in the local identifier's scope.

Gaps

1. The last paragraph of LRM 12.3 is not completely clear as to whether names in generic actual parameters are identified before or after they are substituted for the corresponding generic formal parameters in the specification and body templates. The intent is that generic actual/formal substitution should follow the same static binding rules as nongeneric actual/formal substitution for subprogram calls, discriminant constraints, etc.

12.3.1 Matching Rules for Formal Objects

Semantic Ramifications

Note that a generic formal in parameter is always a copy of the actual parameter, and that a generic formal in out parameter is always a renaming of the actual parameter (see 12.1.1.S). The discussion of "variable_name" and type conversion in 6.4.1.S is relevant here too, except that type conversions of variable names are not permitted as generic actual in out parameters.

Similarly, the sections of 8.5.5 that discuss renaming of variables are relevant here also.

Note that a generic formal in parameter that has an unconstrained array, record, or private type inherits the constraints of the generic actual in parameter (similar to subprogram in parameters, see LRM 6.2).

Compile-time Constraints

1. In a generic instantiation, the base type of a generic actual object parameter must be the same as the base type of the corresponding generic formal object parameter.

2. A generic actual in out parameter must be a variable name. It must not be a type conversion of a variable name.

3. See 12.1.1.C/5.

Exceptions

1. CONSTRAINT_ERROR is raised by a generic instantiation if a generic actual object parameter (or default value, see 12.1.1.G/2) does not satisfy the constraints of the corresponding generic formal object parameter.

Test Objectives and Design Guidelines

1. Check that the base type of a generic actual object parameter cannot differ from the base type of the corresponding generic formal object parameter.

   Implementation Guideline: Check both in and in out parameters.

2. Check that a generic actual in out parameter cannot be:

   - a constant;
   - a parenthesized variable;
   - a function call returning a scalar, array, record, access, or private type value;
   - an attribute;
   - an aggregate, even one consisting only of variables;
   - a qualified expression containing only a variable name;
   - an allocator;
   - an expression containing an operator;
   - a type conversion of a variable.

3. Check 12.3.1.C/3 (see 12.1.1.T/7).

Gaps

1. Many of the gaps in 8.5.G that concern renaming of variables are relevant here with respect to generic in out parameters.

12.3.2 Matching Rules for Formal Private Types

Semantic Ramifications

Note that if a generic formal private type parameter has a discriminant part, the types of the discriminants may depend on previous generic formal type parameters of the same generic part. Hence, the corresponding previous actual type parameters must first be substituted for these formals (similar to LRM 12.3.4 (index and component types in formal array types) and 12.3.6 (parameter and result types in formal subprograms)), after which the discriminant parts of the actual and formal private types can be compared as described in LRM 12.3.2 (see 12.3.2.G/1 and 12.3.2.G/2).

Compile-time Constraints

1. In a generic instantiation, a generic actual unconstrained array type parameter must not be given to a generic formal limited/nonlimited private type parameter.

2. In a generic instantiation, a generic actual type parameter must be a type that has assignment, equality, and inequality available if the actual parameter is given to a generic formal nonlimited private type parameter.

3. If a generic formal limited/nonlimited private type parameter has no discriminant part, then in a generic instantiation, the corresponding actual type must have no discriminants.

4. If a generic formal limited/nonlimited private type parameter has a discriminant part, then in a generic instantiation, the corresponding actual type must have the same discriminant part, i.e., the number and order of discriminants, the discriminant names, base types, and presence/absence of default values must be the same.

Exceptions

1. CONSTRAINT_ERROR is raised by a generic instantiation that has a generic actual type parameter that has a discriminant whose subtype constraint or default value (if any) differs from that of the corresponding generic formal limited/nonlimited private type parameter.

Gaps

1. LRM 12.3.2 is not completely clear about whether both or only one of the bulletted conditions need be satisfied for a successful generic

actual/formal private type match. The intent is that both conditions must be satisfied.

2. It appears to be an oversight that the matching rules in LRM 12.3.2 for generic formal private types with discriminant parts do not first begin by replacing any generic formal type parameters in the formal discriminant part by their corresponding actual type parameters, as is done in LRM 12.3.4 (for index and component types in formal array types) and in LRM 12.3.6 (for parameter and result types in formal subprograms).

12.3.3 Matching Rules for Formal Scalar Types

Compile-time Constraints

1. In a generic instantiation, a generic actual type parameter must be a discrete type (i.e., an enumeration or integer type) if the corresponding generic formal type parameter is defined by a generic type definition of form "(<>)".

2. In a generic instantiation, a generic actual type parameter must be an integer type if the corresponding generic formal type parameter is defined by a generic type definition of form "range <>".

3. In a generic instantiation, a generic actual type parameter must be a floating point type if the corresponding generic formal type parameter is defined by a generic type definition of form "digits <>".

4. In a generic instantiation, a generic actual type parmeter must be a fixed point type if the corresponding generic formal type parameter is defined by a generic type definition of form "delta <>".

12.3.4 Matching Rules for Formal Array Types

Semantic Ramifications

The replacement of generic formal types in a generic array type definition by corresponding generic actual types is part of the completion of the elaboration of a generic part during the elaboration of a generic instantiation, as discussed in 12.3.S.

Compile-time Constraints

1. In a generic instantiation, a generic actual type parameter must be an array type if the corresponding generic formal type parameter is defined by a generic array type definition.

2. In a generic instantiation, a generic actual array type parameter must have the same number of indices (dimensions) as the corresponding generic formal array type parameter.

3. In a generic instantiation, a generic actual array type parameter must have the same component base type as that of the corresponding generic formal array type parameter (after replacing any generic formal types in the formal component type by their corresponding generic actual types).

4. In a generic instantiation, a generic actual array type parameter must, for each index position, have the same index base type as that of the corresponding index position in the corresponding generic formal array type parameter (after replacing any generic formal types in the formal index positions by their corresponding generic actual types).

5. In a generic instantiation, a generic actual array type parameter and its corresponding generic formal array type parameter must either both be unconstrained array types or both be constrained array types (after replacing any generic formal types in the formal index positions by their corresponding generic actual types).

Exceptions

1. CONSTRAINT_ERROR is raised by a generic instantiation for a generic actual array type parameter corresponding to a generic formal array type parameter if the corresponding subtype constraints of the component types differ, or if, for a given index position, the corresponding index subtypes or corresponding index constraints (if any) differ.

12.3.5 Matching Rules for Formal Access Types

Semantic Ramifications

The replacement of generic formal types in a generic access type definition by corresponding generic actual types is part of the completion of the elaboration of a generic part during the elaboration of a generic instantiation, as discussed in 12.3.S.

Compile-time Constraints

1. In a generic instantiation, a generic actual type parameter must be an access type if the corresponding generic formal type parameter is defined by a generic access type definition.

2. In a generic instantiation, a generic actual access type parameter must have the same accessed (i.e., designated object) base type as that of the corresponding generic formal access type parameter (after replacing any generic formal types in the formal accessed type by their corresponding generic actual types).

3. In a generic instantiation, a generic actual access type parameter and its corresponding generic formal access type parameter must either both have unconstrained accessed types or both have constrained accessed types (after replacing any generic formal types in the formal accessed type by their corresponding generic actual types) (see 12.3.3.G/1).

Exceptions

1. CONSTRAINT_ERROR is raised by a generic instantiation for a generic actual access type parameter corresponding to a generic formal access type parameter if the corresponding accessed subtype constraints (if any) differ.

Gaps

1. LRM 12.3.5 says if the formal accessed type is constrained, then the actual accessed type must have the same constraint. If says nothing about whether or not the actual accessed type must be unconstrained when the formal accessed type is unconstrained. However, if a constrained actual access type CAT and an actual variable CAV of type CAT were passed as parameters to an unconstrained formal type UFT and a formal <u>in</u> <u>out</u> object UFO of type UFT, then within the generic unit, one could alter (via assignment) the (index or discriminant) constraint of the object designated by CAV, and thus violate the intent of the language. Note that LRM 12.3.4 requires an actual array type parameter to be unconstrained if and only if the formal parameter is unconstrained. Hence, we believe the intent is that an actual access type parameter must be unconstrained if and only if the formal access type parameter is unconstrained.

12.3.6 Matching Rules for Formal Subprograms

Semantic Ramifications

The binding of a generic actual subprogram parameter to a generic formal subprogram parameter is essentially the same as renaming a subprogram, and hence the discussion of renaming subprograms in 8.5.S is relevant here too.

Compile-time Constraints

1. A generic subprogram must not be used as a generic actual subprogram parameter (i.e., an instantiation should be used instead).

2. See the constraints on renaming subprograms in 8.5.C.

Exceptions

1. See the exception conditions for renaming subprograms in 8.5.E.

Gaps

1. LRM 12.3 says that an entry may be used as a generic actual subprogram parameter, but the matching rule in LRM 12.3.6 only mentions actual subprograms. Since LRM 8.5 allows entries to be renamed as subprograms, the intent is probably to allow entries as generic actual subprogram parameters.

2. See the gaps concerning renaming of subprograms in 8.5.G.

12.3.7 Matching Rules for Actual Derived Types

Semantic Ramifications

The matching rules for a generic actual derived type parameter are those rules that would be applied to a generic actual subtype parameter consisting of the derived type's parent type as further constrained by the derived type's constraints (see 12.3.7.G/1). For example, in

```
generic
    type FI is range <>;
package GP is
    DIV : FI;
end GP;

type DI is new INTEGER range 1..10;

package IP is new GP(DI);
```

the matching rules applied to DI in the instantiation GP(DI) are the same as the rules that would apply to the instantiation GP(PS), where

```
subtype PS is INTEGER range 1..10;
```

But note that it is the actual derived type DI, and not the parent subtype PS, that replaces the generic formal integer type FI in templates for GP to get package IP. Thus, variable IP.DIV is of type DI, and not of type INTEGER.

Compile-time Constraints

1. In a generic instantiation, a generic actual derived type parameter is subject to those compile-time constraints that would apply to a generic actual subtype parameter consisting of the derived type's parent type as further constrained by the derived type's constraints (see 12.3.7.G/1).

Exceptions

1. In a generic instantiation, the matching of a generic actual derived type parameter to a generic formal type parameter raises the same exceptions and for the same reasons (i.e., under the same conditions or run-time constraints) as would the matching of a generic actual subtype parameter consisting of the derived type's parent type as further constrained by the derived type's constraints (see 12.3.7.G/1).

Gaps

1. The word "parent", which was present in earlier draft versions of LRM 12.3.7, is currently missing, apparently by accident since the current rule is a circular definition. The second sentence should read: "On the other hand, an actual type may be a derived type, in which case the matching rules are the same as if its [parent] type were the actual type, subject to any constraints imposed on the derived type."

APPENDIX A

Syntax Cross Reference Listing

This cross reference listing is based on the July 1980 *Ada Reference Manual*. This listing tells where each syntax term is used in the Ada productions, e.g., accept_statement is used in compound_statement and in select_alternative. It also serves as an index showing where each term is defined. For example, the production for accept_statement is defined in section 9.5, page 9-6 of the reference manual. Similarly, the productions for compound_statement and select_alternative are defined in Sections 5.1 (page 5-1) and 9.7.1 (page 9-10), respectively. An ellipsis (...) indicates that a syntactic term is not defined by a syntax production, e.g., see lower_case_letter. The italicized prefixes for some syntactic terms, e.g., variable_name, have been removed for this listing. All uses of parentheses are combined in the term "( )". Reserved words and special symbols appear at the end of the listing.

abort_statement 9.10, 9-14
- simple_statement 5.1, 5-1

accept_statement 9.5, 9-6
- compound_statement 5.1, 5-1
- select_alternative 9.7.1, 9-10

access_type_definition 3.8, 3-29
- generic_type_definition 12.1, 12-1
- type_definition 3.3, 3-5

accuracy_constraint 3.5.6, 3-13
- constraint 3.3, 3-5
- real_type_definition 3.5.6, 3-13

actual_parameter 6.4, 6-5
- parameter_association 6.4, 6-5

actual_parameter_part 6.4, 6-5
- entry_call 9.5, 9-6
- function_call 6.4, 6-5
- procedure_call 6.4, 6-5

adding_operator 4.5, 4-10
- simple_expression 4.4, 4-9

address_specification 13.5, 13-6
- representation_specification 13.1, 13-1

aggregate 4.3, 4-6
- allocator 4.8, 4-22
- enumeration_type_representation 13.3, 13-4
- primary 4.4, 4-9
- qualified_expression 4.7, 4-21

alignment_clause 13.4, 13-5
- record_type_representation 13.4, 13-5

allocator 4.8, 4-22
- primary 4.4, 4-9

argument 2.8, 2-6
- pragma 2.8, 2-6

array_type_definition 3.6, 3-18
- component_declaration 3.7, 3-23
- generic_type_definition 12.1, 12-1
- object_declaration 3.2, 3-2
- type_definition 3.3, 3-5

assignment_statement 5.2, 5-2
- simple_statement 5.1, 5-1

attribute 4.1.4, 4-5
- length_specification 13.2, 13-2
- name 4.1, 4-1

base 2.4.1, 2-4
- based_number 2.4.1, 2-4

based_integer 2.4.1, 2-4
- based_number 2.4.1, 2-4

based_number 2.4.1, 2-4
- numeric_literal 2.4, 2-3

basic_loop 5.5, 5-6
- loop_statement 5.5, 5-6

block 5.6, 5-7
- compound_statement 5.1, 5-1

body 3.9, 3-31
- program_component 3.9, 3-31
- subunit 10.2, 10-6

body_stub 10.2, 10-6
- program_component 3.9, 3-31

case_statement 5.4, 5-5
- compound_statement 5.1, 5-1

(continued on next page)